ROBOT ASSISTED MATERIAL HANDLING
FOR SHIRT COLLAR MANUFACTURING
-- TURNING AND PRESSING --
DLA 900-87-0017 Task 0004

FINAL REPORT

VOLUME VI:

Semi-Autonomous Workstation Control

CENTER FOR ADVANCED MANUFACTURING

CLEMSON
UNIVERSITY
College of Engineering
Clemson, South Carolina 29634

ROBOT ASSISTED MATERIAL HANDLING FOR SHIRT COLLAR MANUFACTURING
-- TURNING AND PRESSING --
DLA 900-87-0017 Task 0004

FINAL REPORT

VOLUME VI:

Semi-Autonomous Workstation Control

Frank W. Paul
Principal Investigator

and

Eric J. Torgerson
Research Assistant

Center for Advanced Manufacturing
and
Clemson Apparel Research

Clemson University
Clemson, SC

June 1992

P00011 to DLA900-87-D-0017

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Unclassified Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Clemson University Clemson Apparel Research 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Defense Personnel Support Center
6c. ADDRESS (City, State, and ZIP Code) 500 Lebanon Road Pendleton, SC 29670	7b. ADDRESS (City, State, and ZIP Code) 2800 South 20th Street P.O. Box 8419 Philadelphia, PA 19101-8419
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Defense Logistics Agency 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DLA 900-87-D-0017 Delivery Order 0004
8c. ADDRESS (City, State, and ZIP Code) Room 4B195 Cameron Station Alexandria, VA 22304-6100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 78011S; PROJECT NO. .; TASK NO.; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Robot Assisted Material Handling for Shirt Collar Manufacturing: Turning and Pressing
Vol. VI: Semi-Autonomous Workstation Control - unclassified

12. PERSONAL AUTHOR(S)
F. W. Paul, Principal Investigator; Eric J. Torgerson, Research Assistant

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM n/a TO _____	1992 June 23	271

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document presents Volume VI of the report of research into the automation of shirt collar manufacturing using robotic methods. Presented is the hierarchical, semi-autonomous control scheme employed to achieve coordinated operation of the various robots and machines employed in the collar turning and pressing workstation. This control scheme was tested and demonstrated through operation of the machine devices discussed in related volumes. Real-world validation of the use of hierarchical and intelligent control in the context of the apparel industry has been demonstrated.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Frank W. Paul	22b. TELEPHONE (Include Area Code) 803-656-3291 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

ACKNOWLEDGMENTS

The authors would like to thank all those who were involved with support of this project. Especially, it is appropriate to thank the Defense Logistics Agency, Department of Defense, for support of this work under contract number DLA 900-87-0017 Task 0004. This work was conducted through Clemson Apparel Research, a facility whose purpose is the advancement of apparel manufacturing technology and by the Center for Advanced Manufacturing, Clemson University.

ABSTRACT

The integration of flexible automation technology to manufacturing systems is impeded by the difficulty of producing the control programs which direct manufacturing operations. These programs link the users of flexible devices (e.g., robots and machine vision systems) to the computing agents responsible for controlling device behavior. The distribution of device activities and control program elements over hierarchical levels and the use of man-machine interface for task planning are regarded as ways of reducing the efforts associated with the programming of manufacturing systems.

Research in intelligent and hierarchical control has approached the development of control structures using conventional techniques such as block diagrams or mathematical formulations. The control of real-world manufacturing systems, however, involves system programming issues such as task planning and inter-device coordination which cannot be easily represented by conventional means. This research takes a software-oriented approach to develop hierarchical control structures which correspond directly to real-world manufacturing activities.

A control and planning scheme is devised for computer-controlled workstations which differentiates device operation issues from program development. Workstation

activities are divided into a three-level hierarchy and are implemented by hierarchical program structures, designed to separate procedural information from hardware-related data. Off-line task-level planning is carried out in the scheme with the assistance of an operator who is familiar with workstation operation. Rule-base systems support the operator by providing task recommendation information and by ensuring plan feasibility. Task-level plans are autonomously converted to control programs which can be downloaded to workstation computers. The scheme has been implemented using PC-based C/Prolog software and has been applied to planning problems involving different robot-assisted assembly workstations.

The contributions of this research are the integration of hierarchical control and operator-assisted task planning to the operation of manufacturing workstations. Control structures are developed which permit the separation of hardware-related data from manufacturing procedures. Planning demonstrations conducted on an apparel manufacturing workstation validate the usefulness of hierarchical and intelligent control theory for real-world applications while identifying research areas which require further study.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF FIGURES

List of Figures (Continued)

List of Figures (Continued)

List of Figures (Continued)

CHAPTER I

INTRODUCTION

Background

The use of flexible automation technology is increasing the productivity of industrial manufacturing. Automated machine devices such as robots and machine vision systems have demonstrated that they can improve quality and enhance productivity. However, one advantage gained by integrating such advanced technology to manufacturing derives from the technology's versatility and flexibility. Robots can be programmed to perform different tasks with minimum machine retooling, while sensors can monitor the manufacturing environment so that system operations can be modified to handle any disturbances. The benefits from flexible automation technology are of greatest value to those manufacturers who frequently alter their production means and who must deal with unstructured environments. A significant problem associated with the application of flexible technology to manufacturing stems from the complicated nature of most manufacturing processes. Whatever flexibility is gained by advanced technology must often be sacrificed in order to make the manufacturing system operational [1].

The solution to dealing effectively with many manufacturing difficulties relates directly to the control of the production apparatus. Conventional control theory has

focused chiefly on performance issues such as improving the response of some system output in tracking a desired input. Research in control has brought about significant improvements in system speed, accuracy, efficiency, and other similar performance measures. Yet conventional control theory has circumvented many of the larger control issues critical to manufacturing automation. These issues include task planning, system monitoring, man-machine interface, and error handling. Control theory also tends to divorce itself from the means by which control is implemented which for devices such as robots and flexible machines involves the use of a programmable computer.

The computing capabilities for typical flexible automation equipment are often too sophisticated for the small and medium sized manufacturer. The lack of any standard programming language for robot and sensor systems currently on the market compounds the manufacturer's difficulties [2]. In most cases, the physical capabilities of automation equipment is not at issue. Robots or other flexible machines can handle the manufacturing tasks assigned to them if they are supplied with the proper control programs. Developing these control programs quickly and reliably is one principal challenge for manufacturers interested in automation. The demand for software, however, exceeds the ability to produce it, and furthermore, the software that is often created may not do what the user needs or wants [3].

Researchers such as Bourne and Fox [4] are taking a more global approach in the study of manufacturing control. Factory-wide issues such as job scheduling, material ordering and transporting, and resource allocation are being investigated with regards to how they impact the direction of manufacturing tasks. A critical subcomponent of any factory incorporating flexible technology is the entity known as the manufacturing workstation. A robot-assisted manufacturing workstation typically includes a single robot manipulator situated adjacent to the machines it is to service. Objects are brought into the workstation, are manipulated by the robot, have operations performed upon them by the other workstation machines, and (when all such operations are completed) are removed from the workstation. Control issues at a workstation-level include designating and sequencing of machine activities, interdevice communication, status monitoring of the processed objects, and fault detection. Easing the programming burden and simplifying the control of automated workstations are the primary intentions of this research.

The Defense Logistics Agency (DLA), a division of the Department of Defense responsible for acquiring non-weaponry items for the military, is interested in enhancing the productivity of uniform manufacturing. One research project funded by this agency is to develop and implement a robot-assisted workstation which is capable of performing critical

manufacturing tasks on shirt collars. These tasks are currently achieved by a skilled manual operator with the aid of a single machine. The robot-assisted workstation developed for the project is used to demonstrate the control concepts and practices developed in this research.

The shirt collars are constructed from two trapezoidal-shaped fabric plies sewn together on three sides as illustrated in Figure 1.1. The plies are stitched together with their outer surfaces positioned on the inside; inverting or "turning" the collar is performed to reorient the outer ply surfaces properly and to fold the stitch line inside the collar pocket where it cannot be seen. The edges of the collar assembly are creased through a pressing operation. Turning and pressing shirt collars involve intricate manipulations of the workpiece; the operator must possess sufficient dexterity and eye-hand coordination to place each collar properly in the turning and pressing machine. Figure 1.2(a) shows a front view of the machine which the operator utilizes to turn and press shirt collars while Figure 1.2(b) depicts the operator turning a collar on the machine. Only skilled operators can produce turned and pressed collars of acceptable quality in a consistent and repeatable manner.

An Apparel Assembly Workstation (AAW) has been developed for automating shirt collar turning and pressing. Special turning and pressing devices have been devised which facilitate the loading of single collars by means of a robot

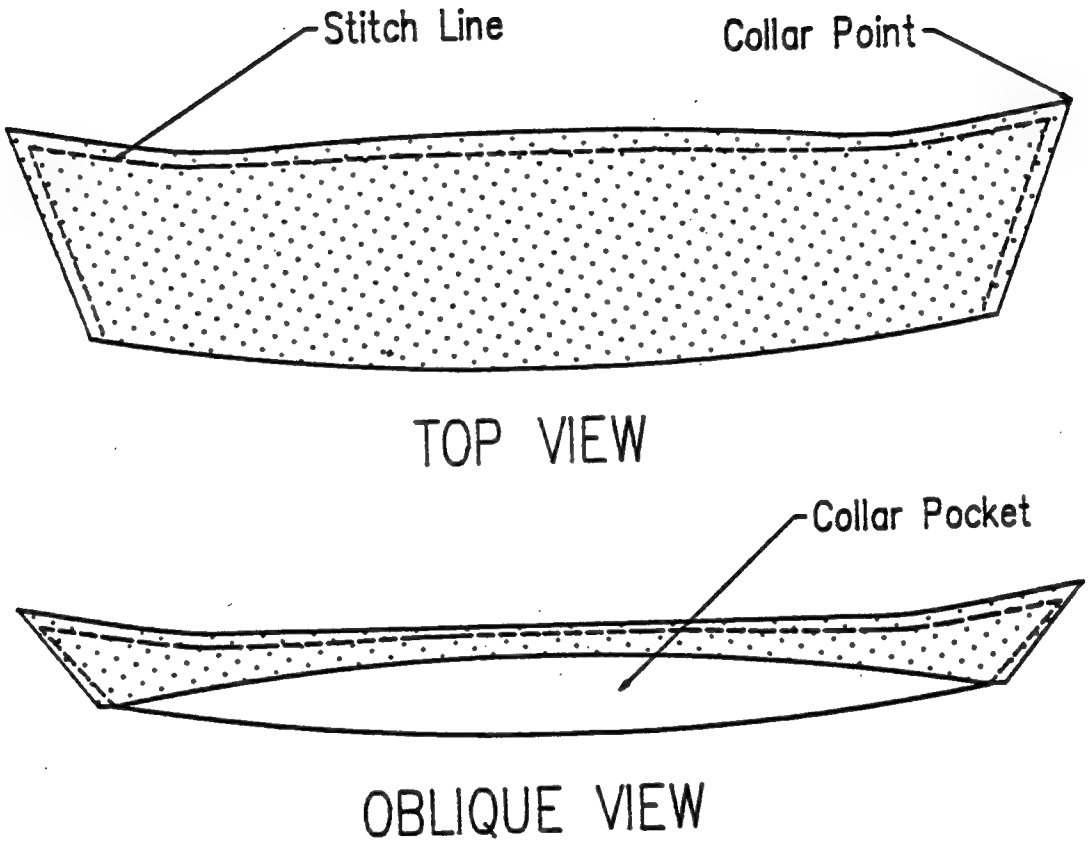
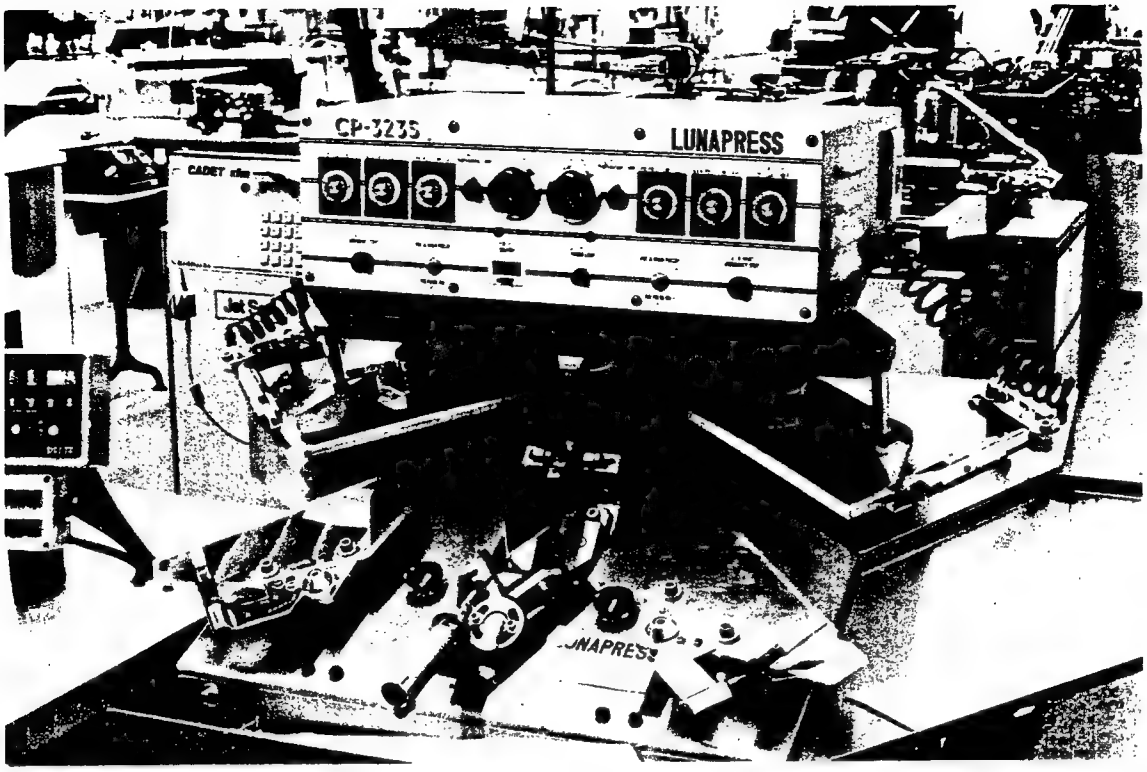
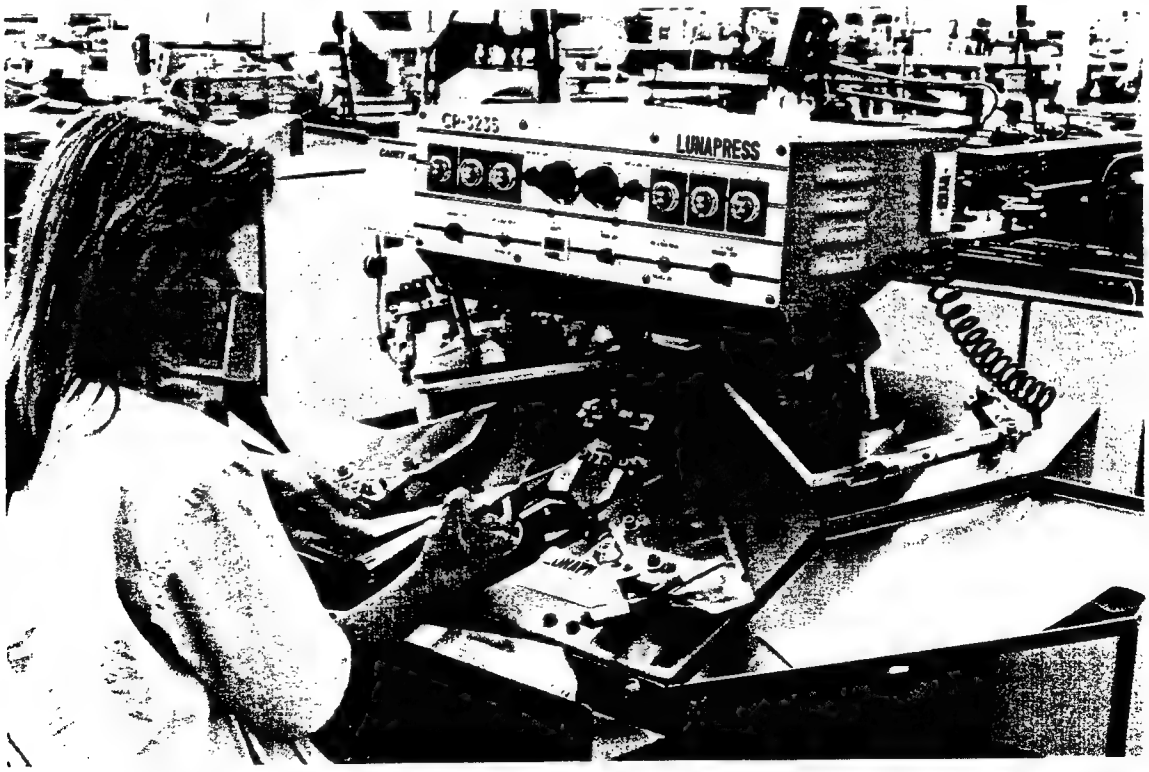


Figure 1.1. Unturned Shirt Collar



(a) Front View of Machine



(b) Manual Turning Operation

Figure 1.2. Collar Turning and Pressing Machine

with machine vision assistance. The AAW is controlled by PC-level computers, selected over more powerful computers because of their low cost and widespread familiarity. A scheme for controlling and designating AAW operations is needed which permits a human operator (not a programmer) who is knowledgeable of AAW task capabilities to manage the creation of workstation control programs.

Problem Definition

The control of manufacturing workstations which utilize flexible automation technology falls beyond the boundaries of conventional control methodologies. Organizing workstation components and activities to simplify workstation control would contribute to the development of manufacturing automation. The fusion of man-machine interface with hierarchical control is viewed as one approach to achieving this goal. The objectives of this research concerning workstation control are presented to frame the problem and to introduce the proposed solution. The contribution of the research to the present knowledge base is also considered.

Research Objectives

The goal of this research is the development of a new hierarchical semi-autonomous control scheme which conforms to the operational capabilities of a robotic workstation directed by multiple PCs. The first research objective to address in this goal is a greater understanding of how system configurations and operations are represented. The

control of system hardware is to be connected hierarchically from high-level program modules to low-level functions which direct elementary machine processes. Another objective concerns the development of operator-assisted planning of workstation activities. Methods are sought which distribute, in an appropriate manner, the numerous duties associated with the creation of workstation control programs to a single workstation operator, system programmer(s), and operator-assisted computing agents. The coordination and execution of workstation tasks, once formulated into a plan, is to be achieved with minimal human intervention.

The implementation and demonstration of the proposed hierarchical control scheme comprise the final research objectives. The scheme will initially be applied to a hypothetical manufacturing job involving the assembly of objects into a finished product. The creation of control programs for a robot-assisted workstation (the AAW) will verify the scheme's usefulness to real-world manufacturing. The scheme simplifies the modification of control programs to variations in workstation configuration and/or job specification. Complementary issues such as implementing error recovery procedures and determining how Artificial Intelligence (AI) can assist in workstation planning are also examined. Techniques used in this research derive from information theory and its integration with conventional feedback control formulations.

Knowledge Contribution

An enhanced understanding of how to represent and to organize computers and devices within a system at a workstation-level is viewed as one contribution of the research. The integration of hierarchical control strategies to a robotic workstation should reveal much about the utility of dividing system control structures into levels of varied purpose and scope. Defining more precisely the roles that workstation operators, skilled programmers, and computing facilities should play in the creation of control programs is perceived as another important result of the proposed work. Knowledge will be gained on the management and manipulation of information relevant to workstation control. The development of a specific hierarchical semi-autonomous scheme will provide new techniques for enhancing the utility of a manufacturing workstation by reducing the time and effort associated with its programming and reprogramming. This work is seen as an important bridge between control research at the factory-level and conventional control theory which concentrates on low-level hardware concerns such as equipment performance.

Dissertation Content

An extensive literature review is provided in Chapter II. This review examines hierarchical and intelligent control theory and its usefulness in the control of complex systems such as manufacturing workstations. Literature

devoted to activity planning with and without human participation is studied. Real-world manufacturing issues are investigated with emphasis placed on apparel assembly research. The chapter concludes with literature on the nature of distributed systems and on strategies for dealing with system errors.

Chapter III details the control and planning concepts upon which the workstation control scheme is based. The means by which systems and system activities are represented and classified is presented. The reasons for organizing control software into a hierarchy are explained. The chapter also includes discussions on the planning scheme and the way that human input is combined with autonomous computer actions for plan generation. The role that AI rule-base systems play in assisting with job planning is defined.

Discussion concerning the implementation of the proposed control scheme can be found in Chapter IV. The C/Prolog software package, termed ODCAP for Operator-Driven Controller And Planner, which demonstrates the control scheme is reviewed in detail. The methods by which workstation hardware is classified and modeled are examined. The planning algorithm which constitutes the primary feature of the ODCAP software is presented with the use of flowcharts and pseudocode. Chapter V provides an example of job planning using ODCAP. The planning example concerns a hypothetical assembly problem involving a robot-assisted workstation.

Chapter VI describes how ODCAP is utilized to plan jobs for a real-world manufacturing workstation, the AAW. The composition and operation of the AAW is briefly detailed. The applicability of ODCAP-generated plans is validated by utilizing them to generate control programs which command actual AAW tasks. Chapter VII presents conclusions and recommendations associated with the developed scheme's contribution to new knowledge and methods for its improvement and expansion.

CHAPTER II

LITERATURE REVIEW

Introduction

Considerable research has been conducted on the various topics related to workstation control and job planning. The review begins with insight on hierarchical and intelligent control, both of which are considered "higher level" than conventional closed-loop feedback control. The principal idea behind these types of control is the division of the controller into different levels with higher levels possessing more "intelligence" but less resolution than their lower counterparts. Literature pertaining to the programming and planning of activities is covered from AI sources as well as robot language sources. Many specific implementations of planners and programming methods are examined. The planning review is divided into investigations of schemes not involving (autonomous) and requiring (human-assisted) human interaction.

Any study of workstation control should consider real-world implementation issues for integrating flexible technology to manufacturing. Similarly, since a workstation is a distributed system of computing agents and devices, literature sources pertaining to distributed systems concepts are examined. The final topic covered in the literature

review is error recovery which concerns the alteration of system behavior to cope with unexpected occurrences.

Hierarchical Control Literature

Albus [5] defines hierarchical control as a vertical stack of control levels connected by command and feedback information which pass between the levels. Each level has components for sensing system behavior (feedback), for executing control commands (actuation), and for comparing perceived system behavior with the expected (modeling). The sensing and command components alter and determine, respectively, the state of the physical system while the third component takes information from the other two and places it in proper context by relating it to known experience. The levels differ in their degree of generality (scope) and their degree of sophistication (intelligence); higher levels have greater intelligence but are not as specific.

The principle upon which intelligent control (the terms hierarchical and intelligent control are considered interchangeable) is founded is that of increasing intelligence with decreasing precision as stated by Saridis [6]. He believes an intelligent system should have the ability to interact with humans (man-machine interface), to coordinate control actions, and to interact with the environment via sensory feedback. Saridis [7] considers a system intelligent if it can perform decision-making and planning, providing an excellent review of existing control methodologies

and the evolution of the intelligent control concept. Another important precept for hierarchical control is that of problem reduction, the decomposition of complex problems into smaller, more manageable pieces.

Meystel [8] views intelligent control as the fusion of AI, Operations Research, and conventional control theory. Closed-loop control schemes are often inadequate for large systems because their system descriptions are limited to formulations expressed in terms of differential/integral and difference calculi. Intelligent control places control theory into a more global scheme, one which is concerned with more than just closed-loop performance. The author partitions intelligent control into three fundamental levels (listed from top to bottom): the organization level, the coordination level, and the hardware control level. The organization level interprets input commands and defines system tasks for the lower levels. The middle level receives commands from above and coordinates task execution as performed by the hardware control level.

Applications of hierarchical control schemes to physical systems are not common. One substantial example is the Automated Manufacturing Research Facility (AMRF) at the National Bureau of Standards (currently the National Institute of Standards and Technology) as detailed by Norcross [9] and Albus et al. [10]. An elaborate hierarchical control system is utilized to automate the actions of a factory microcosm. AMRF control is classified into five levels: (1) facility,

(2) shop, (3) cell, (4) workstation, and (5) equipment. CAD/CAM and management information systems all reside at the top. Feedback and command data filters up and down, respectively, through the different levels. One notable difference between the levels is the lengths of their respective planning horizons, defined as the time elapsed between control commands and actions. Lengths vary from weeks and months at the facility level to hours and minutes at the workstation level to seconds and milliseconds at the hardware level. Despite these time disparities, the lowest-level actions are fully coordinated with their highest-level counterparts. At the workstation level, determining the relationships between devices and the objects they act upon is considered paramount. The control strategy for the AMRF is completely data-driven. Every component on every level receives information, processes it, and passes it to to another component. Data is not only analytic in content but procedural as well.

Saridis [7] discusses the implementation of intelligent control to Stanford University's autonomous robot vehicle known as "Shakey". The control scheme directs the vehicle's movement in an unknown environment by sensing obstacles and planning appropriate actions. The use of hierarchical control structures to improve the performance of robot joint actuation is presented by Acar [11].

Planning Literature

Meystel [8] perceives planning as a topic falling within the domain of AI and not within that of conventional control. Planning is defined by Schalkoff [12] as determining a course of actions which propels a system from some initial state to a final, goal state. Planning is very important at the workstation-level where hardware and software structures may already exist for performing specific tasks, but it is unclear how to arrange these structures to achieve some high-level goal. The review separates planning literature into autonomous and human-assisted divisions. Autonomous planning systems still require human input at some level and in some form, as indicated by Brooks [13]; the difference between the two divisions is primarily one of degree. The manual writing of a program in some computer language is viewed as a form of human-assisted planning.

Xia and Bekey [14] make the distinction between planning and scheduling. They view planning as the off-line, intentional setting of some course of system actions while scheduling implies the on-line, opportunistic application of system resources to the planned actions. This research views scheduling as a component of planning and not as a

separate entity. The review focuses on planning issues over scheduling ones.

Autonomous Planning

A comprehensive study of existing autonomous planning systems has been done by Gevarter [15]. He reduces planning strategies into four principal categories: (1) non-hierarchical, (2) hierarchical, (3) skeletal, and (4) opportunistic. Non-hierarchical planners create only one plan representation while hierarchical systems generate more than one, each possessing a distinct level of abstraction (higher-levels are typically more abstract). Skeletal planners initially cast the plan in a rough outline form. Once this "skeletal" plan is devised, the planner reviews it and inserts the required details. Opportunistic planning systems behave in the same manner as humans. The plan is divided into sections which are subsequently developed when the opportunity arises. The author provides many examples of operational planning systems, most of which are implemented in an AI language such as LISP or Prolog.

Hutchinson and Kak [16] advocate the following planning objectives, listed from highest to lowest priority:

1. the achievement of operational goals,
2. the satisfaction of geometric constraints, and
3. the reduction of uncertainty.

A thorough examination of how information should be manipulated for the control of engineering systems has been

conducted by Lu [17]. He believes that engineering systems are "information-rich" but "knowledge-sparse" and suggests that AI should be used as a problem solving tool and not as the one solution. The term "information" refers to the data which defines a system's composition and behavior while "knowledge" implies the utilization of this data for the performance of desired system actions. Gevarter [18] reviews current AI techniques and methodologies and demonstrates their usage.

Examples of automatic planning schemes are numerous; the ones mentioned in this review all concern robot planning. The schemes are typically applied to artificial problems involving the robotic manipulation of blocks (referred to as "blocks-world" problems). The work of Prajoux et al. [19] affords a perfect example of such problems. Paul, Durrant-Whyte, and Mintz [20] and Pang [21] utilize an AI construct known as a blackboard to carry out planning. The blackboard behaves as an executive deciding which of the planning solutions posted to it by various knowledge sources (Paul et al. [20] refers to these elements as sensing, action, and reasoning agents) will be implemented by the system. Blackboard approaches are inherently opportunistic. A robot planning system known as SHARP has been developed by Laugier and Pertin-Troccaz [22]. The authors divide robot activity into three primary tasks: grasping, transporting, and part-mating. SHARP makes use of kinematic and dynamic models of the robot and its environment and the objects that

the robot manipulates. The planning system generates actual computer code (in the language AML) for directing the actions of an IBM robot.

Another planning system, labeled SROMA, generates assembly sequences for the automated construction of objects from their component parts [23]. SROMA makes use of AI constructs known as frames to hold information on the objects and their method of assembly. The off-line, workcell planner named ESSOP employs an expert system to perform automatic path planning, sensor management, error recovery, and kinematic and dynamic simulation [24]. A graphical user interface is required to configure the workcell layout and to initialize the planning problem.

A robotic workstation planning system called SPAR combines domain independent techniques with programming modules which are task specific [16]. Every action planned by SPAR possesses a list of preconditions (conditions which must be existent if the action is to be applied) and lists of conditions which are added to and deleted from the knowledge base if the action is performed. Dufay and Latombe [25] utilize inductive learning for their planning scheme. The learning is carried out in two phases, the teaching phase and the induction phase. The first phase involves the repeated performance of some single task with variations occurring in task execution due to environmental disturbances. A general set of procedures for achieving the task is inferred in the

induction phase by "merging" the actions performed in each repetition into a single plan.

Human-Assisted Planning

A thorough study of human decision-making and man-machine interface has been carried out by Sheridan and Ferrell [26]. They suggest that human decision-making is based on perception, implying that it is easier for a person to show a system how to operate than to program its operation without seeing any intermediate results. Groover [27] lists four techniques by which humans program robot actions: (1) manual setup, (2) teach method, (3) computer-like languages, and (4) off-line programming with graphics simulation. The first two methods involve direct human interaction with the robot to teach it the actual motions it must make to perform the desired task. Programming robots with computer languages (such as Adept's "V+") requires that the human operator have programming knowledge. This assumption may not be true in many small and medium size manufacturing settings. The final technique affords the user a friendly, interactive environment in which to designate robot activities.

Taylor et al. [28] discuss the need for creating programming interfaces between the workstation and its operator which account for the operator's programming skill. They suggest a menu-driven interface for operators unfamiliar with conventional programming languages. Lozano-Perez [29]

reviews many of the various robot programming languages currently on the market. He divides them into two basic groups: task-level languages and robot-level languages. Task-level languages require user specification of operations based on their effects upon objects. The user deals with specific robot actions when programming with robot-level languages. The vast majority of languages are incorporated into the second group. Languages at a task-level are inherently more complex and require geometric models of the robot and its environment. The author believes that most programming languages are inadequate and recommends that robot languages should be cast as supersets of standard programming languages such as "C" which include powerful computing structures.

An excellent example of a human-assisted robot planning system, termed XPROBE, can be found in the work of Summers and Grossman [30]. They propose that it is unrealistic to expect non-programmers to devise elaborate computer code utilizing geometric data which is not readily available. XPROBE is a menu-driven system which continually updates sensor and other status parameters and displays them to the user. Computer code is generated automatically once the user has concluded the planning session. XPROBE behaves like a sophisticated teach-pendant which accepts commands on a task-level.

Rembold [31] employed hierarchical concepts for programming robot tasks. He has created a sophisticated

language called SRL which makes use of an expert system and graphics display to assist its user with programming. Shen and Wong [32] place an implicit programming layer (which is supplied with tasks by a user) above an explicit layer of conventional robot programming languages. The implicit layer divides the tasks it is given into specific commands for the conventional languages. Some of these commands might lead to actions not explicitly requested by the user. A sensor programming language has been devised by Johnson and Hill [33] involving instructions which continually monitor sensory conditions. Execution of the tasks commanded by these instructions is not completed until the sensory conditions are considered favorable.

Additional Literature Review

In order to control and to plan workstation manufacturing tasks, a control scheme must reflect real-world issues. The review of manufacturing considerations examines basic research on improving manufacturing as well as focused research on apparel assembly automation. Research pertaining to the integration of device components into a distributed system is considered. Literature related to the automation of error handling for improving the robustness of system behavior is also investigated.

Real-World Manufacturing

Bourne and Fox [4] propose that decision-making on the shop floor must be automated to make manufacturing more

autonomous and flexible. They state that flexible technology has the drawbacks of increasing the complexity and the amount of required decision-making. The authors list the following items as the main features of shop-level manufacturing:

1. a set of predefined parts,
2. one (or more) operation sequences defined for each part,
3. workstations where the operations are performed,
4. orders and materials which enter the shop, and
5. finished products which exit the shop.

Two excellent examples of robot/vision workstations controlled by PC-level computers which carry out complex manufacturing operations are provided in publications by Hill, Burgess, and Pugh [34] and Osada [35].

The concept of integrating product design with the product's method of manufacture is advanced by Nevins and Whitney [36]. Past practice was to treat design and manufacturing as separate, somewhat unrelated entities. The authors provide a comprehensive list of design factors which influence manufacturing. Boothroyd, Poli, and Murch [37] contend that product designers must work to simplify methods related to product assembly (and disassembly) and part positioning and feeding. Many product design features are detailed which achieve these criteria.

The integration of flexible automation technology to apparel manufacturing lags behind that of other industries.

The problem of low fabric stiffness and the lack of capital investment have impeded efforts toward this integration [38]. The transport and handling of materials are the most time consuming jobs in the sewing industries [39]. Taylor et al. [40] provide a complete study of the areas of apparel manufacturing which need to be automated. Descriptions of robotic systems designed to implement apparel manufacturing tasks are provided in these publications [41-46].

Distributed Systems

Compton [1] proposes that communication within a distributed system is typically handled in an ad hoc fashion. Separate programs must be written for each computing element with complex real-time control issues such as system communication performed by explicit programming instructions. The author suggests combining all the separate programs into a single one with system control details (such as communication) kept hidden from the user. He also believes that control software should be collected into interchangeable modular units which can be linked to different hardware configurations, in the same manner that hardware components are often grouped into modules. Fitzgerald and Barbera [47] use low-level control interfaces to modularize their distributed robotic system.

The hardware configurations feasible for distributed systems are reviewed by Shin and Epstein [48]. The two most common types of configurations are termed loosely-coupled

and tightly-coupled. In a loosely-coupled system none of the system components dominate the entire group; message passing occurs only when components request information from other components. Communication in a loosely-coupled system often delays the execution of other system actions. Tightly-coupled arrangements place one system component in a master-slave relationship with the other components. Harmon [49] contends that loosely-coupled systems should use local area networks to implement communication while a single high-speed bus is recommended for tightly-coupled arrangements.

An example of a sophisticated distributed robotic system which utilizes Sun computers and a VME bus is described by Stewart, Schmitz, and Khosla [50]. The system makes use of hierarchical control software which keeps real-time structures, such as interrupt handlers, transparent from the user. Jones, Barkmeyer, and Davis [51] relate that implementing hierarchical control on distributed systems may be difficult because such control might conceal relationships between levels not apparent in an examination of the hierarchy. They also reason that an existent, inefficient hierarchy might already be embedded in the system and will

typically be used as the foundation of any developed control scheme.

Error Recovery

Fielding, DiCesare, and Goldbogen [52] suggest that 80% of all programming effort is dedicated to specifying error recovery procedures. They advance that system programs be augmented with additional commands for error recovery. Various error recovery strategies are described by the authors. Lam, Pollard, and Desai [53] reduce errors to three types: exceptional, hardware malfunction, and plan error. An exceptional error has no apparent explanation and cannot be predicted. Plan errors are due to either poor reasoning (an example would be a robot colliding with another device) or uncertainty (an example of this would be the misalignment of a robotic hand with some object it is grasping). Smith and Gini [54] contend that an error recovery system should attempt to get plan execution back on course before attempting more elaborate recovery procedures.

Norcross [9] defines four steps associated with error handling: (1) isolation, (2) investigation, (3) correction, and (4) continuation. Cox and Gehani [55] classify error recovery procedures as either backward or forward. Backward procedures transform the system to a previous state and resume system operations when conditions are more favorable. Forward strategies involve the system's direct handling of the error without delay. Brooks [13] points out that the

robot, the objects it manipulates, and the environment it operates in all possess uncertainty which can lead to errors.

A few researchers have applied error recovery concepts to improving the robustness of automation. Gini, Gini, and Somalvico [56] have developed a sequential two-part scheme for error recovery. Error handling is first achieved by an error-independent activity and then by an error-dependent activity. The second activity is the selection of a specific error recovery procedure based partly on the generic information produced by the first activity. Taylor and Taylor [57] used probability information in a matrix format to establish connections between error causes and actual sensor readings. They desire the elimination of continual, time-consuming monitoring of system sensors for error detection. Brooks [13] has developed an automatic plan checker which makes use of CAD-based data. The checker propagates plan errors by substituting symbolic parameters for dimensional uncertainties and is able to verify whether a given plan will work, given all the geometric constraints.

CHAPTER III

CONTROL AND PLANNING CONCEPTS

Introduction

The management and control of a workstation in the performance of manufacturing tasks involves issues, such as motion trajectory planning and process scheduling, which are not addressed in conventional control theory. Sharir [58] reduces these issues to three broad categories:

1. **sensing** - determining a world model,
2. **planning** - using the world model to command actions, and
3. **control** - executing and regulating planned actions.

Fielding, DiCesare, and Goldbogen [52] contend that the information needed to carry out these items is usually available but must be altered or "transmuted" from an unusable form to a useful one. Sheridan and Ferrell [26] believe that human operators should be assisted by computers in supervising and directing complex machine activities because humans have a "limited capacity to handle information".

Program algorithms, referred to as software, are the means by which human operators utilize computers to command and control system behavior. The drawbacks associated with software are its lengthy development time and the need for skilled programmers to produce it. The programming of workstations controlled by multiple computers entails difficult

issues such as system communication and process synchronization. Techniques are needed for separating the development of software (programming) from the designation of workstation tasks (planning). This separation would benefit an operator who is familiar with the tasks he/she wants to command a workstation to perform but is unable to create the necessary program code. The removal of job planning concerns from the software would also ease the burden of the programmer who develops the code.

Hierarchical control permits the formation of a single plan for a distributed system of agents by combining low-level plan instructions into commands involving (possibly) the entire system. Off-line semi-autonomous planning means a human operator can command workstation activities and view (in simulation) the effects of those activities on the workstation's environment without involving the actual workstation hardware. Integrating hierarchical control concepts into off-line semi-autonomous planning should reduce the problems associated with workstation management by distributing them, in an appropriate manner, among the workstation operator, the software programmer(s), and the computing agents.

This chapter first addresses the different ways in which systems are represented and the means by which hardware and software can be differentiated in system representations. The arrangement of program elements, along with the activities they command, into a hierarchy is presented.

Program elements such as instructions and data are placed into strict syntactical forms. Human-assisted off-line planning of workstation activities is examined with specific roles proposed for human operators and programmers as well as computing agents.

System Representation

A system's configuration and utility can be depicted in a variety of manners. Several methods of representation are required to depict the distinguishing features of a particular system. For example, the representation of a workstation's control software should be different from a representation of the workstation's hardware configuration. The way in which a system is depicted will often affect how it is controlled. Systems, such as manufacturing workstations, which cannot easily be modeled with algebraic equations and differential/integral calculus must have representations which simplify and define the means by which they are controlled.

Conventional methods for representing system behavior (and its control) are initially reviewed. A representation which separates system control issues from physical system elements (hardware) is proposed. Workstation hardware and computing agents are arranged into a particular configuration which is conducive to the control and planning concepts proposed in this chapter.

Conventional Methods

Three types of representations are frequently encountered within control and computer theory: (1) the Algorithmic State Machine (ASM) diagram, (2) the block diagram, and (3) the job hierarchy chart. Each type portrays some aspect of the system that it represents with greater clarity than its two counterparts. The ASM diagram depicts system "states" (the status of the system at one particular time) and the transitions between the states. Figure 3.1 includes an ASM diagram illustrating the transitions of water between its natural states of solid, liquid, and vapor. The circles denote system states while the arrowed lines signify the transitions. The means by which transitions occur (the addition/removal of heat for this example) are not presented in conventional ASM diagrams. ASM diagrams are best utilized to describe system behavior without showing how this behavior is controlled and directed to the achievement of some desired goal.

Block diagrams are the customary means by which control systems are represented. These diagrams depict the interaction of control signals with various system elements. The system elements, termed "blocks", denote analytic formulations such as algebraic or differential/integral equations which depend upon the signals for input and output. A block diagram showing the closed-loop feedback control of some dynamic system is included in Figure 3.2. The disadvantages associated with block diagrams stem from their inability to

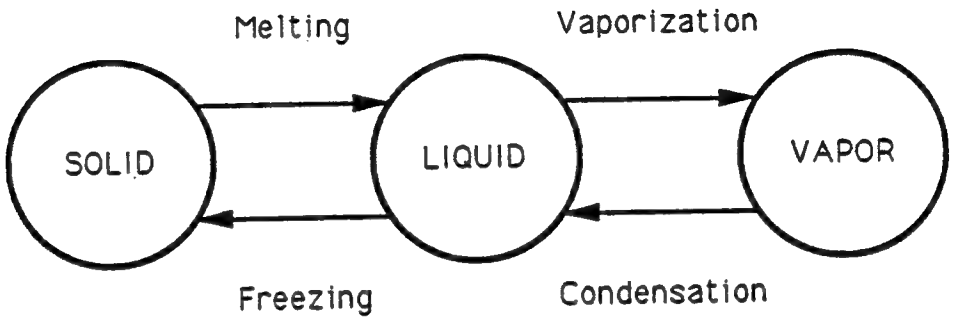


Figure 3.1. ASM Diagram for States of Water

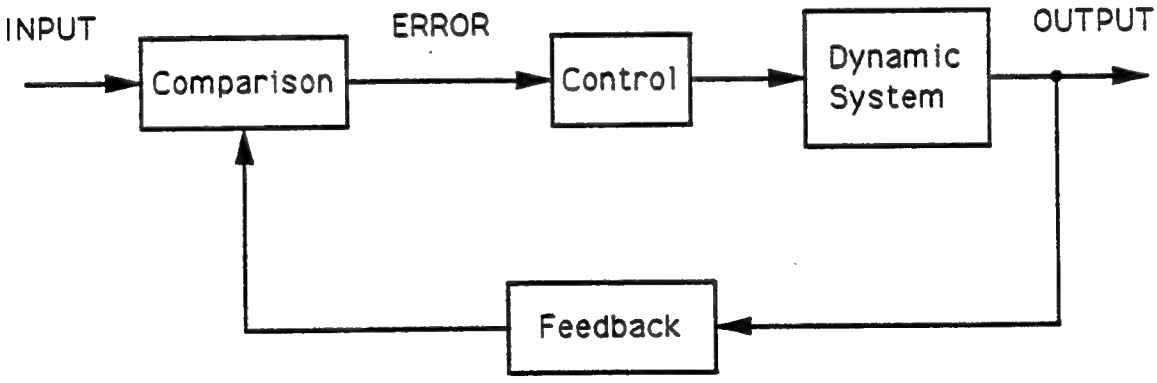


Figure 3.2. Block Diagram for Dynamic System Controller

depict clearly the status of the system and the execution sequence of system actions. Block diagrams, however, provide an excellent means of representation for systems which can be modeled completely with algebraic and differential/integral expressions. The control of individual actuators or sensors is best portrayed with block diagrams since the signals and elements of the diagrams corresponds approximately to the signal voltages (or currents) and electrical (or mechanical) components respectively of the physical system.

The activities that complex systems can perform are typically represented with the use of a job hierarchy chart. Figure 3.3 shows a sample chart composed of three hierarchical levels with activities labeled (in descending order) as jobs, tasks, and processes. The labels are selected arbitrarily but are indicative of the division and subdivision of system activity. The example chart does not suggest an execution order for system activities; it illustrates how system operations are reduced to sets of smaller, constituent suboperations. The "command flow" of a system is demonstrated by such hierarchical charts, meaning that the performance of a job (task) is contingent upon the commanded execution of certain tasks (processes). As with ASM diagrams, hierarchy charts do not reveal how system activities are achieved.

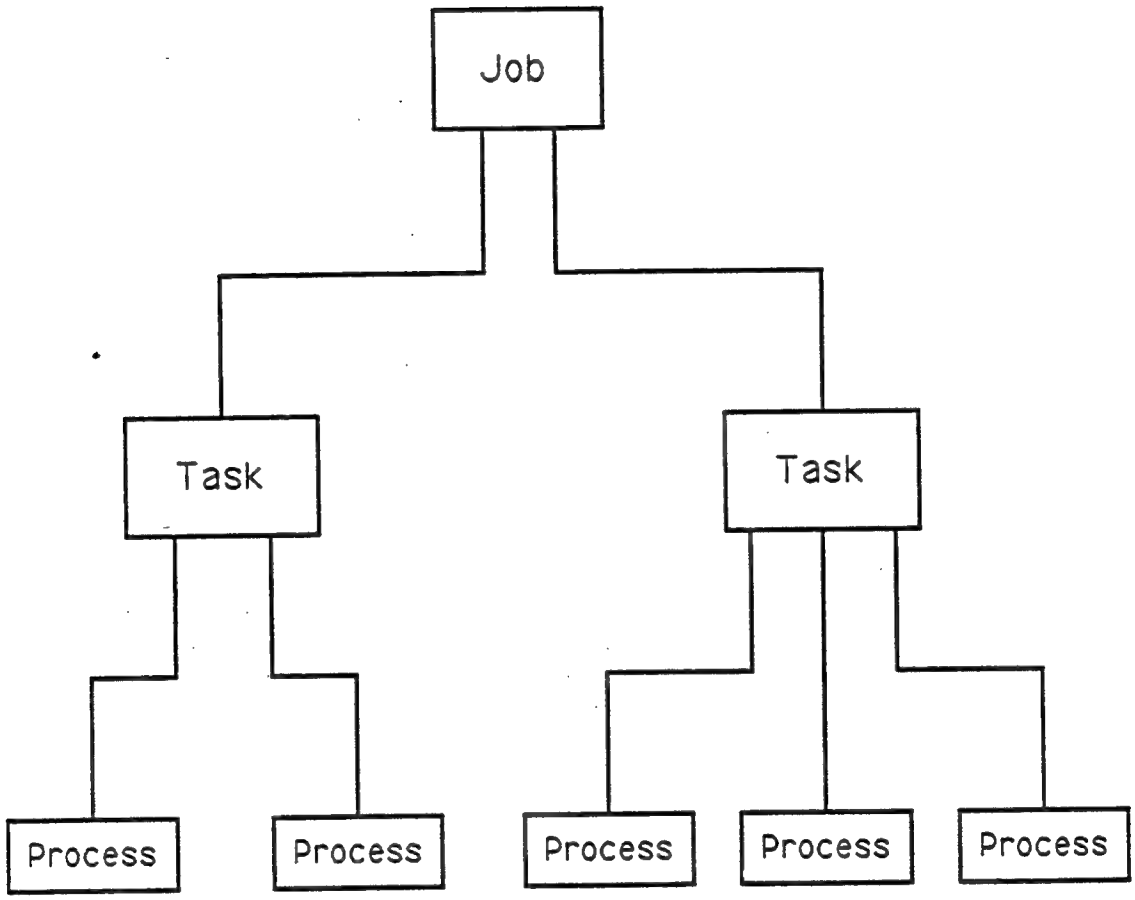


Figure 3.3. Job Hierarchy Chart

Hardware and Software Differentiation

Automated workstations consist, in general, of computer-controlled devices which act upon objects to transform them into useful products. The computer-controlled activities performed by workstation devices are best represented by hierarchical charts such as the one in Figure 3.3. The actual operation of devices and their interaction with objects, however, are more accurately depicted with block diagrams. An integrated representation combining these two methods is proposed. Figure 3.4 shows a generic control scheme where software control aspects (involving "command flow") are separated from hardware control (involving "signal flow").

All system devices (some exceptions are considered in Chapter IV) are viewed as either actuators or sensors, instruments for effecting or measuring, respectively, changes to the workstation environment. Devices are depicted as blocks in Figure 3.4 while objects are represented as circles (a single actuator, sensor, and object are shown in the figure for clarity). Computers are not considered devices since they do not directly affect or sense the environment. Dashed line connections between the actuator, sensor, and object indicate the indirect relationships existent between these items. A vision sensor, for instance, does not come in contact with the object that it is sensing. An object's state can be altered by an actuator, but the same actuator performing the same activity can also alter a different

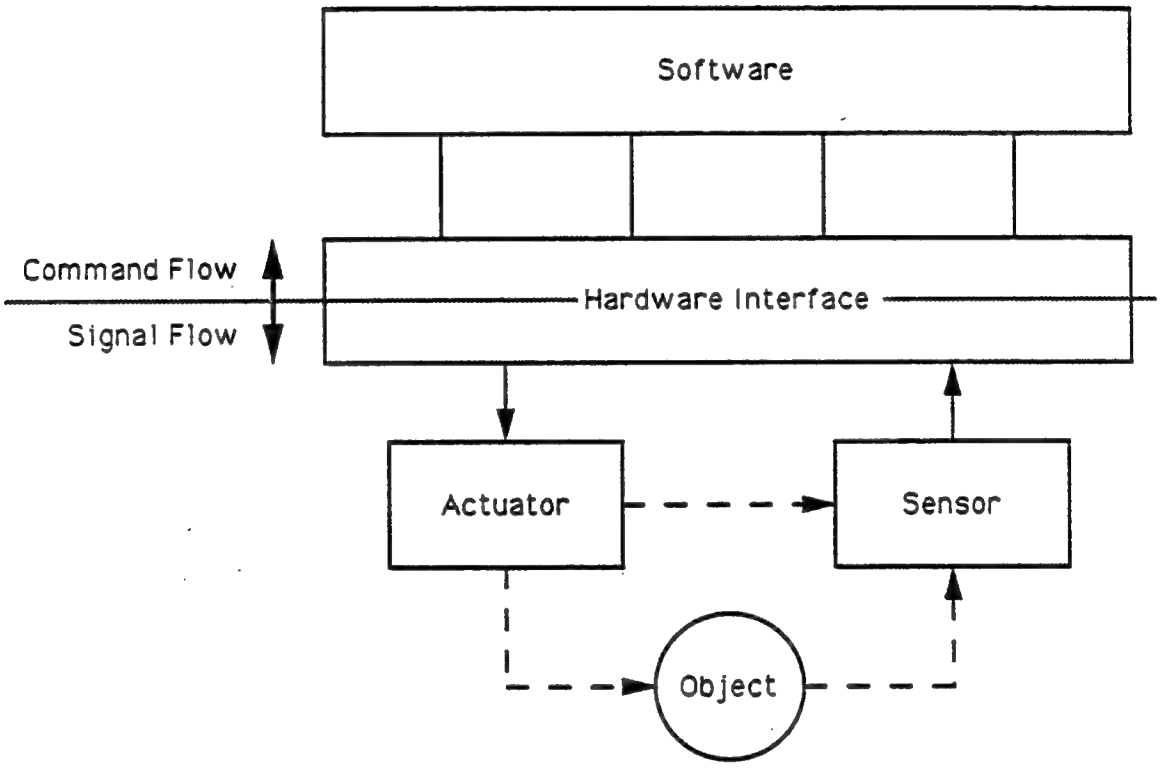


Figure 3.4. Generic Control Scheme

object in a different manner. The solid arrowed lines in Figure 3.4 signify the control and data signals which link devices to their computer controllers. The hardware interface (a system element incorporating electrical equipment such as signal conditioners and amplifiers) serves as the bridge between software commands and device signals and is considered as part of the computer systems. Software is represented as a single block in the scheme but will be subsequently expanded into a hierarchy.

In conventional control schemes, actuators are usually integrated with sensors such that the sensor provides the feedback signal for the closed-loop control of the actuator. Representation of closed-loop control is possible with the proposed combination scheme. Two schemes are proposed, one which involves computer control and one which does not. A local feedback scheme, as shown in Figure 3.5, incorporates a sensor/actuator pair linked together by a feedback signal. The computer provides input to the actuator and receives output from the sensor but does not use the sensor output to determine the actuator input.

Another means for representing closed-loop control is termed the independent feedback scheme. In this scheme, computer processing of the sensor feedback is required to obtain the control signal for the actuator. This scheme, illustrated in Figure 3.6, is considered "independent" because the software responsible for generating the control signal from the feedback is kept hidden (not available for

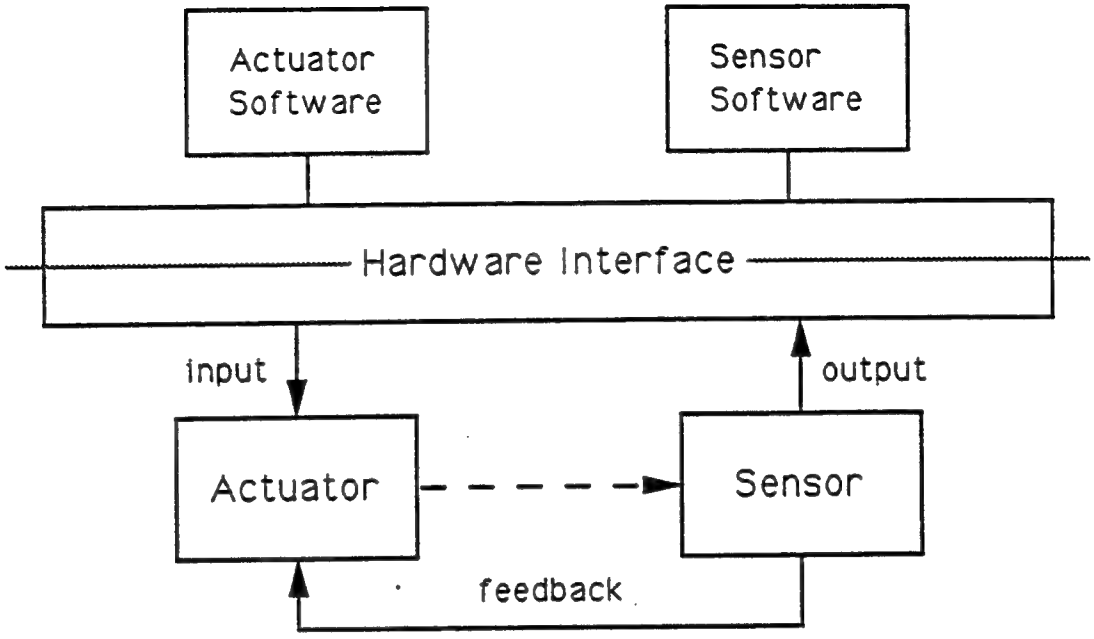


Figure 3.5. Local Feedback Scheme

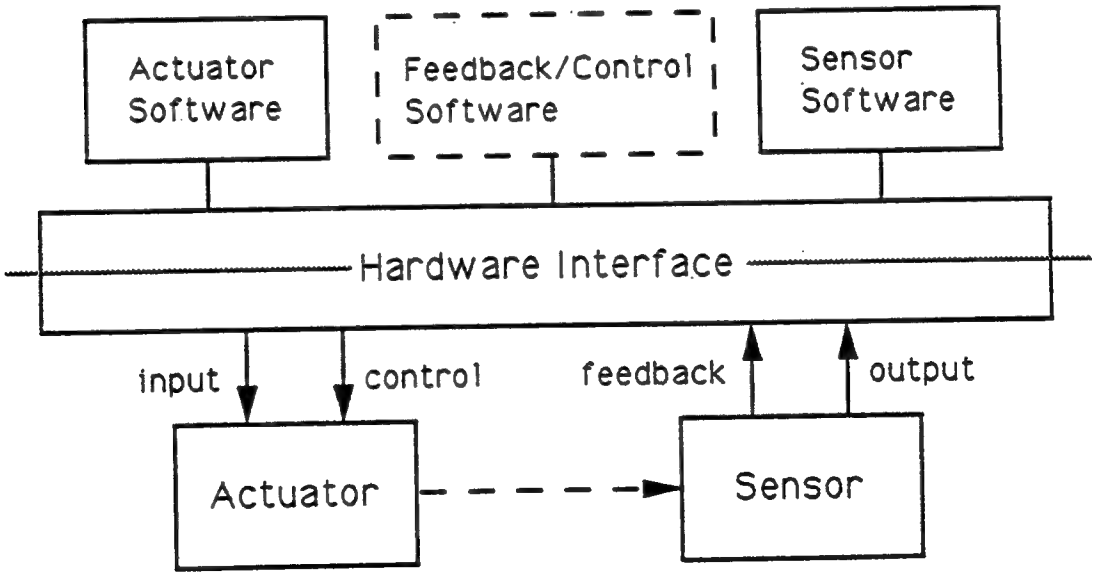


Figure 3.6. Independent Feedback Scheme

modification, as indicated by the dashed lines enclosing the feedback/control software block) from any system user. Independent feedback control schemes are common in robot systems with "closed" computer architectures where users can affect where the robot moves but cannot alter the feedback control strategies which govern how the robot moves. The differentiation of command and control issues enables system representation to be indicative of the actual operation of system elements as diverse as software and hardware.

Workstation Organization

Typical manufacturing workstations require the computational resources that a small number of PC-level computers could provide. A workstation configuration is proposed which integrates PC-level computers with workstation devices such that a distributed system is created. The proposed configuration, depicted in Figure 3.7, includes (n + 1) computers all labeled Controllers except for one computer termed the System Supervisor. The System Supervisor is elevated to a central role in workstation control and should be viewed as two computers (physically it is just one), one termed Controller S and the other known as the Supervisor, because of the dual role that the System Supervisor performs. The Supervisor serves as a junction point for all system communication whereas Controller S behaves exactly as

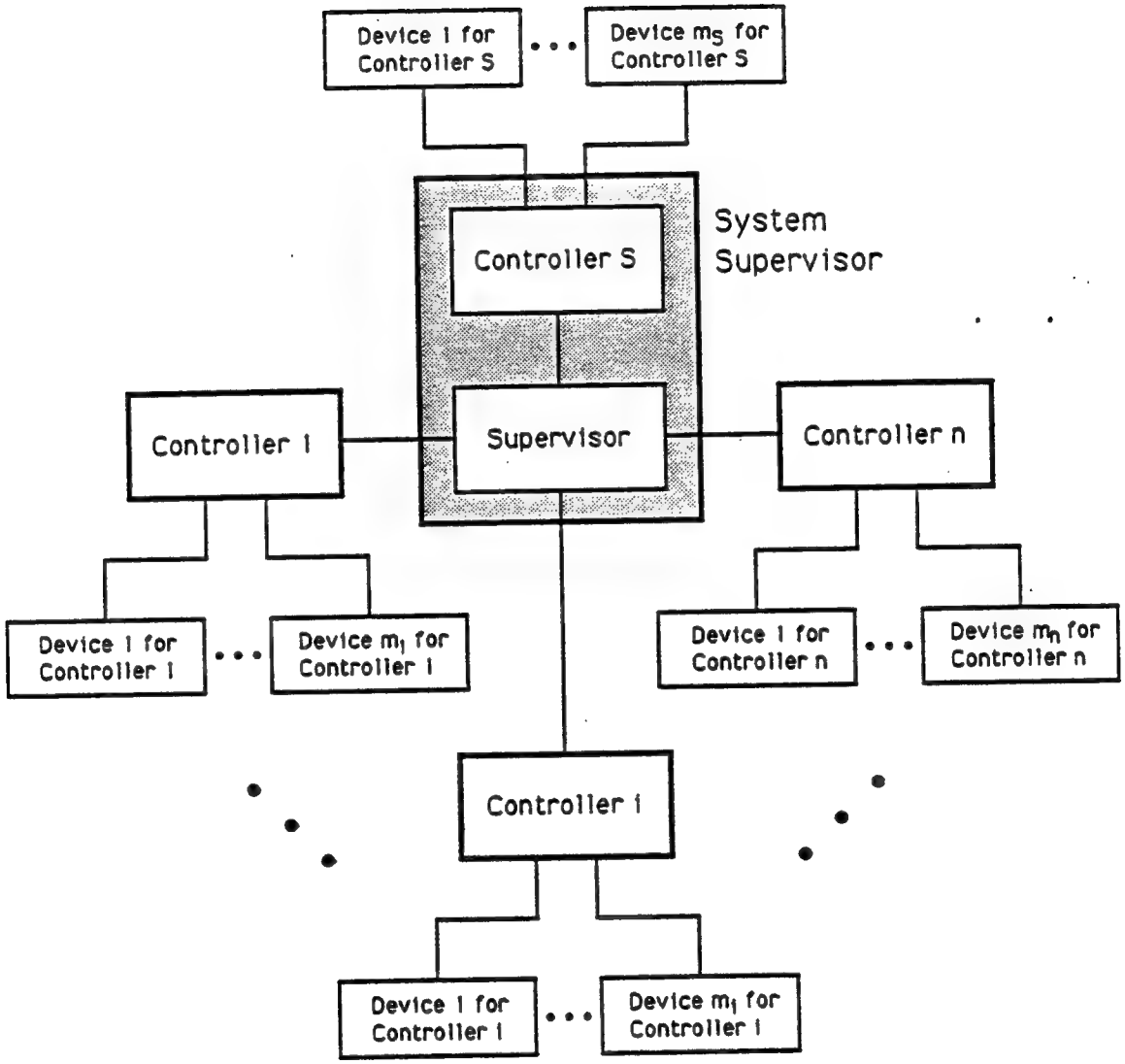


Figure 3.7. Workstation Configuration

any other Controller in the configuration. PC-level computers typically possess the computing power needed for both roles.

Attached to every Controller i (where i denotes either a value between 1 and n inclusive or the letter S) are m_i devices which the Controller directs exclusively. Devices, as stated previously, comprise either actuators (such as robots) or sensors (such as cameras). All computing equipment including that of the hardware interface is encompassed in the n Controllers and the System Supervisor. Workstation resources are arranged in a "star" configuration about one central computer in order to preserve flexibility. Devices can be added or removed from Controllers and, similarly, Controllers can be appended or eliminated from the system without significantly altering the workstation's control configuration. Hardware considerations such as communication rates and the complexity and number of workstation devices will determine the value of n; typical workstations have n values ranging from two to five.

Control Hierarchy

The designation and control of activities for a workstation incorporating multiple devices and computing agents involves complex control elements known as software. The arrangement of these elements into a hierarchy is viewed as one means of coping with the inherent complexities. A hierarchy simplifies the creation of control software by

reducing it into small, manageable operations and by dividing system control into several levels. These levels must address the control issues unique to their place in the hierarchy. For instance, low-level control software will be tied more closely to actual device operation than its high-level counterpart. Software structures and activity divisions are needed which reflect the control hierarchy.

All systems, no matter how autonomous they become, require human input at some level. For systems with little autonomy, this input may be the specification of individual machine operations combined with the data essential to their execution. Autonomous systems need, at the very least, identification of high-level job goals. Consequently, an important issue for workstation control is the delegation of responsibility between the three creators of system control structures: the expert programmer(s), the workstation operator, and the computing resources (acting autonomously). A definitive control hierarchy aids in delegating the duties that each of these agents should perform.

Activity and Software Division

The software block of Figure 3.4 is expanded into a three-level hierarchy with upper levels possessing a broader extent of workstation control than lower ones. The software hierarchy is depicted in Figure 3.8 above the usual hardware elements in the generic control scheme. The hierarchy is composed of three software entities, one for each level,

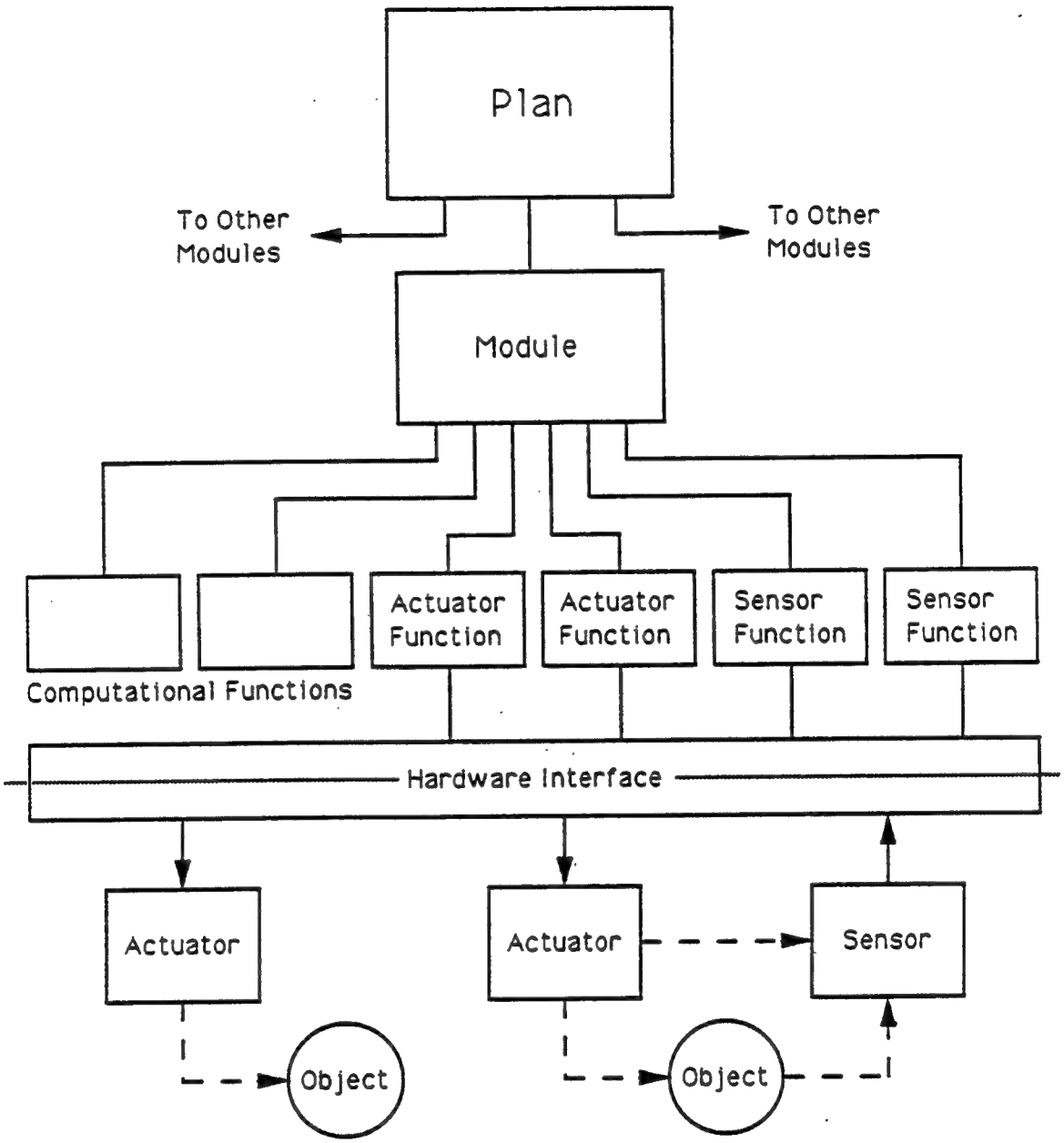


Figure 3.8. Generic Control Scheme with Hierarchical Software

labeled (from top to bottom) as the following: plan, module, and function. These entities correspond exactly with the activity divisions in the job hierarchy chart of Figure 3.3. The labels for the software entities are chosen arbitrarily as were those of their activity counterparts. The scope and utility of the software entities are defined along with their relationships to one another.

A plan is a complete program entity composed of software instructions and the data needed to execute the instructions. Whenever it is desired that the workstation perform a particular job, a single distinct plan must be created. Plans are not linked together as are the other software entities; new workstation jobs require the development of new plans. Each plan entails the use of any number of software modules, as indicated in Figure 3.8. A module incorporates control software which resides in and directs the actions of any number of workstation computers and devices respectively. Modules are responsible for carrying out tasks which typically involve the interaction of workstation devices with objects. Ramamritham and Arbib [59] state that tasks should be designated by their effects on objects rather than by the device actions performed to achieve those effects. Modules must be sequenced such that all the tasks connected with the plan's job are executed in the proper order.

Functions represent the "building blocks" of control software and, unlike modules, are specific to a single

workstation computer. Many of the processes that functions command are related closely to device-specific actions. Functions direct such diverse activities as controlling actuator motion, reading data from sensors, performing numerical computations, and relaying information from one computer to another. Every module consists of a set of function sequences (one for each control computer) whereas each plan is a single sequence of modules. A significant difference between a plan and a module is that the former is defined chiefly in terms of job goals while the latter is defined in terms of specific actions. Figure 3.9 illustrates how the activity and software entities are constituted, and also indicates the parallelism between hierarchical activities and software elements. Tasks and processes are mapped directly to corresponding modules and functions (i.e. Task 1 is performed when Module 1 is executed).

The individual tasks (or processes) within a particular task (or process) sequence all involve a single set of workstation devices and objects. The control of workstations with different hardware configurations yet identical activity sequences must take into account the hardware differences. If a workstation job concerns the automated assembly of some product from individual components, all the assembly tasks in the job must consider specific component features such as geometry and method of assembly. Software entities are not coupled by shared hardware but by shared information. If a device within a workstation performs a

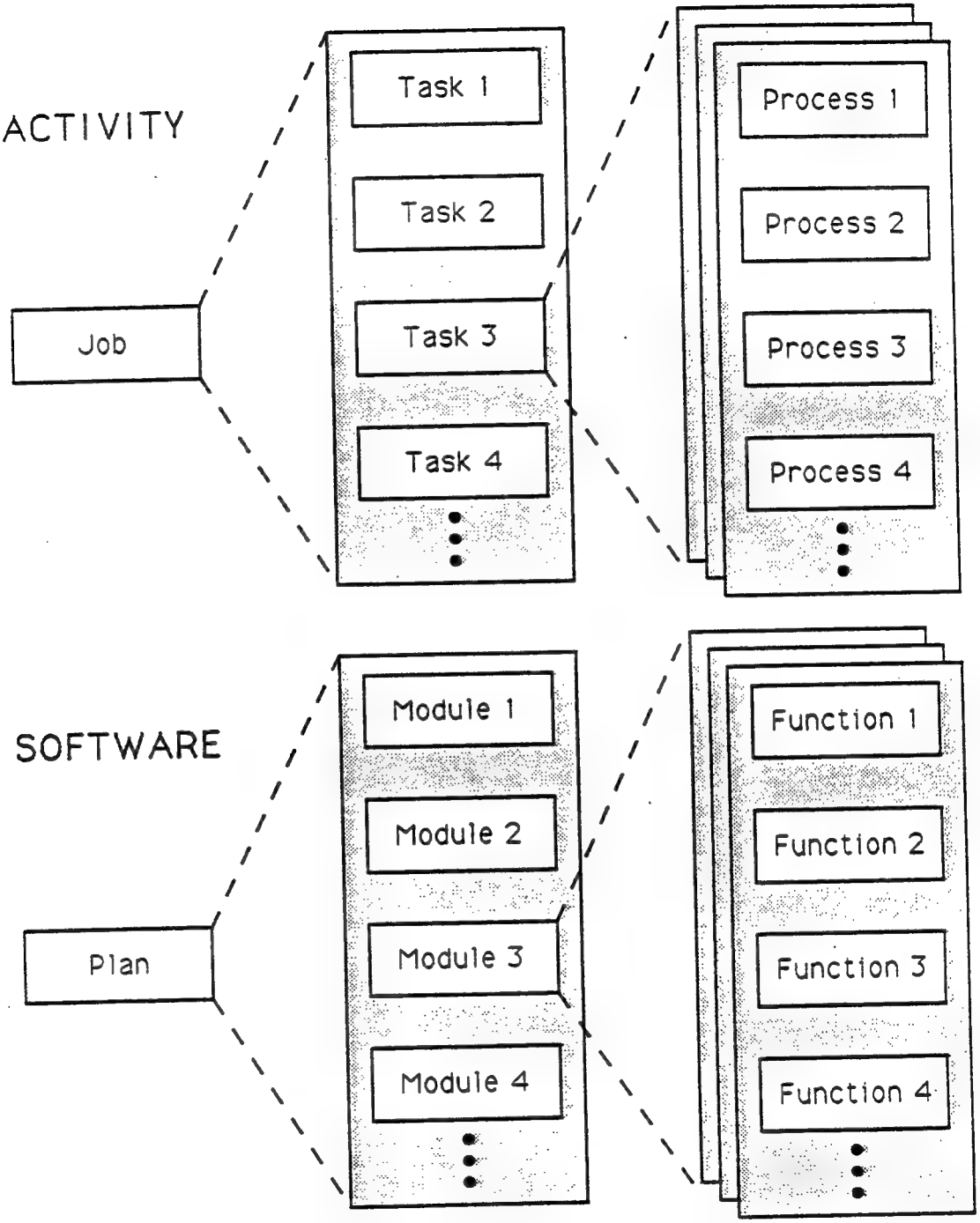


Figure 3.9. Activity and Software Parallelism

certain task, it may be replaced by another device which behaves in an identical manner. The alteration of the task sequence will involve the substitution of the first device with the second. The modification of the control software will require the replacement of information pertaining to the first device with that of the second.

The separation of data from the procedure is critical to the utility of any modular programming scheme. The means to achieve this separation is by grouping information unique to each workstation device and object. Data should be classified in a manner which is compatible with the hierarchy of procedural elements shown in Figure 3.8. For example, low-level data such as the output signal of a device should be linked to the process-level while high-level information such as the fact that a particular device is acting on a particular object is appropriate to the task-level. The next section of this chapter creates software structures which implement the proposed hierarchy of control procedures with suitable data structures.

Plan Operators and Data Structures

Tasks are classified (modules have the same classifications) into three different types, defined as follows.

action: An activity where some physical change in the workstation environment is caused by some device actuator and/or where some environment parameter is "sensed" by some device sensor.

computational: An activity where no device actuators or sensors are affected but where computational analyses are performed.

conditional: An activity where the task execution sequence is altered based on the results of some computational analyses.

Processes (and functions) possess the same three classification types plus two additional ones, defined as follows.

communication: An activity where information is exchanged between workstation computers.

synchronization: An activity where at least two workstation computers are synchronized.

A task's type depends primarily upon the types of all its constituent processes. If a task contains just one action process, the task is viewed as type action. If a task includes no action processes but does alter the task execution sequence, the task is classified as conditional. Tasks which do not satisfy either of these two qualifications are deemed computational.

Every plan is composed of two parts; one part is referred to as the introductory data section while the other is the listing of the plan's constituent modules. The introductory data section consists of all the information the plan needs in order to execute properly. An example of introductory data would be the geometric information associated with the joining together of two objects by an assembly workstation. Introductory data can be obtained from the workstation operator and from data bases. The means by which introductory data is determined and is utilized are described in the planning example of Chapter V.

The module listing of a plan is henceforth termed the Master Plan. Every module in a Master Plan possesses the following syntactical form:

```
module_name
({list of FINITE-STATEs},
 {list of input variable-structures})
({list of output variable-structures})
```

where items in the first pair of parentheses indicate the module's input and, similarly, items in the second pair correspond to the output. Functions are specified (when they are listed within modules) using a nearly identical syntax.

```
function_name
({list of FINITE-STATEs},
 {list of input variable-structures})
({list of output variable-structures})
```

When modules and functions are mentioned throughout this dissertation, they will be expressed as **module_name()()** and function_name()(), respectively, with module names boldfaced to differentiate them from function names. Action and computational module types are embodied in single modules while conditional types require sets of two or more modules.

The double sets of parentheses are adopted to explicitly show the flow of information into and out of the module (or function). Plan information is held in three possible data structures (two of which appear in the module's and function's syntax). Data structures known as FINITE-STATEs (capitalized to signify that their values are designated from finite sets) refer to data elements which are selected during planning. Three kinds of FINITE-STATEs exist at a

task-level, defined as follows: devices, objects, and assemblies (assemblies are defined as groups of connected objects). Process-level FINITE-STATEs include two additional kinds termed the "switch label" and the "switch state" both of which are defined in the "modeling" section of the following chapter.

FINITE-STATEs should not be viewed as input in the conventional sense but as a means for separating data from procedure. The formation of a module or function which can perform the same activities using different devices and objects is made possible with FINITE-STATEs. A programmer can, for instance, create a module which executes a task involving the acquisition and placement of an object by a robot in which device- and object-specific data is omitted from the procedure. A FINITE-STATE specifying a particular robot and one identifying a certain object would serve to indicate what device- and object-specific information need be obtained for the module.

Variable-structures represent groups of related system variables (may be just one) arranged in a predefined manner. The numerical content of these system variables is not determined until the module (or function) in which they are located is executed. There are two possible syntactical forms for variable-structures at a task-level (within modules), defined as follows:

name:FINITE-STATE:COMPUTER

name:COMPUTER

The "name" of a task-level (or process-level) variable-structure can never change for a given module (or function); consequently, the first form must be applied when system variables are linked to device- or object-specific information, resulting in the necessity to identify the particular device or object associated with the variable-structure. An example variable-structure requiring the first syntactical form would be one consisting of a set of joint variable values for a robot where the robot is indicated by its FINITE-STATE label. Even though only one Master Plan is developed (at a time) for a workstation controlled by (n + 1) computers, variable-structures are not retained (necessarily) during plan execution in the memory of every one of those computers but (normally) reside in that of a single computer. The specification of a single resident computer is required for both task-level syntactical forms. Process-level variable-structures (which are contained in functions) do not need identification of either a particular FINITE-STATE or computer due to the fact that a function is always executed by a single computer and a process-level variable-structure's name can be changed to any value.

The third type of data structure is provided the descriptive label of transparent. Transparent data structures are not considered as module (or function) input or output but are data sets incorporated into a function's program code. Transparent data is the data which cannot be separated from procedure. The specification of transparent

data structures is carried out by the programmer who creates the function software code.

Modules and functions can possess any number of FINITE-STATEs and variable-structures in their input and output data lists. If a module incorporates a device or object in a variable-structure, a FINITE-STATE representing that device's or object's name must be included in the module's input list. Identical variable-structures can be listed in both the input and output of the same module; this duplication indicates that the module requires the variable-structure's data and may alter that data. Within a module (or function sequence) input variable-structures must have identical counterparts contained in the output lists of modules (or functions) which are located at an earlier point in the module (or function) sequence, before the sequence is considered viable. This condition implements the means by which software procedures are joined together, namely the coordination of information flowing into and out of the procedures. Determining the sequence in which modules should be placed to create a useful job plan is called planning and is the focal point for the remainder of this chapter.

Planning Considerations

The complexity of planning workstation jobs depends on the device/object interaction involved in performing those jobs. Most workstations utilize equipment specifically

designed and configured to handle all the various duties assigned to them. Jobs typically consist of the transfer of objects (by robots or by human operators) into the workstation, the performance of assembly or other manufacturing tasks upon those objects, and the removal of the finished product made from the objects. The planning associated with such jobs is generally not difficult at a task-level; plans can be assembled by an operator who is familiar with the capabilities of the workstation and who can command task-level actions. The coordination and execution of planned actions below a task-level, however, may be very arduous involving (possibly) the computation of robot trajectories, the analysis of sensory data, and the determination of communication parameters.

A scheme is desired which gives a workstation operator the means to conduct job planning at a task-level while insulating him/her from job coordination and execution issues below a task-level. A skilled programmer will still be needed to organize the software into modules and functions, but he/she is to be unburdened of all task-level planning concerns. The software can be created for use with a variety of devices and objects in opposition to having device- and object-specific information embedded in the code, which limits the software to one hardware configuration. A three-tiered control scheme is presented whose highest level consists of task-level planning (off-line) and whose lower levels carry out the coordination and execution (on-line) of

the plan generated at the topmost level. The levels associated with this scheme should not be confused with those of the activity and software hierarchies. Figure 3.10 illustrates the proposed workstation control scheme with the three control levels labeled (from top to bottom) as follows: the Planning Level, the Coordination Level, and the Hardware Level. The function and scope (as well as the labeling) of the levels are similar to those of Meystel [8]. The elements which comprise the Planning Level are discussed in the next section; the purpose of this level is the generation of control software for the lower levels.

The one task-level plan generated at the Planning Level is divided into (n + 1) process-level subplans which, in turn, must be passed to the n workstation Controllers and to the System Supervisor. The Coordination Level is made up of these subplans once they have been activated. The subplans communicate with one another through the physical communication links depicted in Figure 3.7 on page 40. Elements residing at the Coordination and Hardware Levels correspond exactly with the entities shown in Figure 3.4 on page 36 for the generic control scheme. Each workstation device is controlled by a single computer and subplan. The final three sections of Chapter III focus on the composition and operation of the Planning Level. The appropriate handling of system errors is viewed as an additional planning issue. A means for dealing with this issue at the Planning Level is presented.

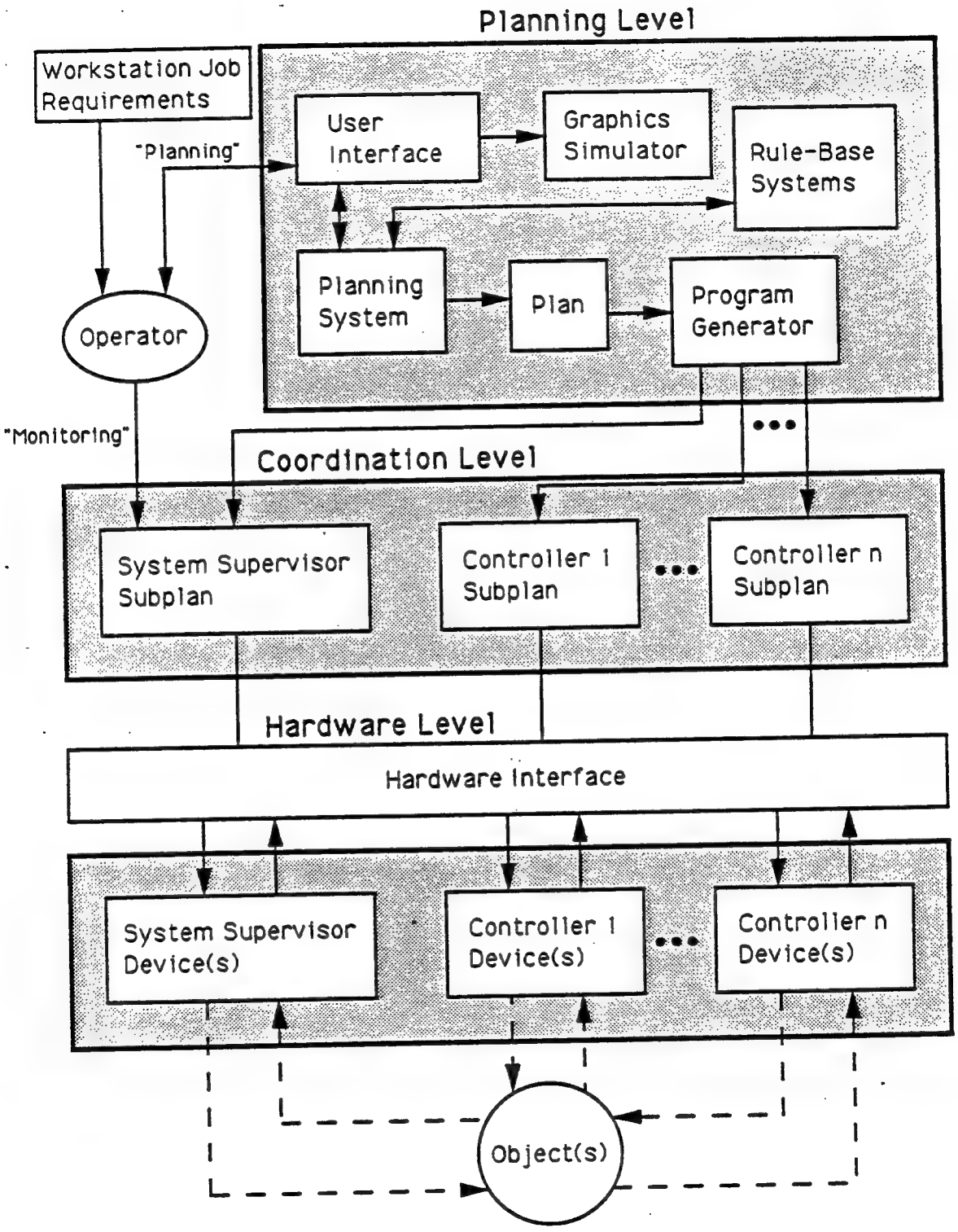


Figure 3.10. Workstation Control Scheme

Semi-Autonomous Off-Line Planning

Planning a workstation job requires interaction between a human operator and the computers which implement the planning scheme, implying that the scheme is semi-autonomous at the Planning Level. The planning computers can never command any physical workstation activity since all Planning Level elements are considered off-line. The workstation operator, who possesses knowledge regarding what tasks the workstation devices are capable of, communicates with the planning system by means of a user interface. The user interface assists the operator with the decision-making aspects of job planning. A graphics simulator provides the operator a simulated view of workstation devices and objects. The simulation is beneficial because it demonstrates visually for the operator the effects that planned activities would have upon the workstation if the plan was implemented on-line.

Each plan is constructed by the operator who in turn is assisted by the planning system. Rule-base systems (these systems are detailed in the next section) situated at the Planning Level analyze the workstation status and use this analysis to recommend to the operator which task can and should be applied to the task sequence. Once the user selects a task, the corresponding module associated with that task is inserted into the plan at the proper location within the module listing (Master Plan). Operator-selected

modules are added to the Master Plan in a strictly sequential manner. The addition of conditional modules, however, can alter the execution sequence of the modules into one different from that of the module listing. The effects of conditional modules upon planning are explained by example in Appendix A.

Certain modules can be inserted into the Master Plan by the planning system without consulting the workstation operator. These modules are termed implicit; modules which are recommended by the planning system and selected by the operator are referred to as explicit modules. All conditional and action modules are classified as explicit because alteration of the workstation environment and/or the plan execution sequence should only be performed with the operator's consent. Implicit modules are always computational in type and usually involve the retrieving of information from the plan's introductory data section as well as the relaying of information between workstation computers.

At the conclusion of a planning session, the planning system can assemble automatically (n + 1) process-level subplans composed of the function listings pertaining to every module included in the Master Plan. Appendix B provides a detailed example (involving a pick-and-place task) which illustrates the division of a module into function listings. The subplans can subsequently be converted by the program generator into control programs, created using the programming language of the Controller or System Supervisor they

are to control on-line. Conversion only requires that changes be made in the subplan's syntax since subplan functions are already written at a process-level. Chapter VI addresses subplan conversion by example. On-line execution of the plan can begin after the converted subplans are downloaded to their respective workstation computers and compiled into executable code.

Additional planning system features permit the operator to modify the workstation data connected to an existent plan and to "replan" the tasks performed by an existent module sequence (Master Plan). Modification of a plan's input data is referred to as "re-simulation" and does not involve changing the plan's module sequence, but it does necessitate that the plan be executed off-line ("re-simulated") so that the introductory data section can be reconfigured with updated workstation information. The augmentation of a finished plan is viewed as an error recovery technique and is addressed in the last section of this chapter.

Rule-Base Contribution

Two distinct rule-base systems, one termed explicit and the other implicit, are utilized at the Planning Level. The Explicit Rule-Base (ERB) system assists the workstation operator by providing him/her recommendations on what module can be applied to a given plan to achieve a desired result. The Implicit Rule-Base (IRB) system operates autonomously,

inserting modules into the Master Plan that make it workable. Both systems consist of collections of "rules" organized into rule-bases and groups of facts held in data bases. Rules when combined with facts provide a qualitative method for deducing new system information based on existent conditions. Rules in AI systems (such as expert systems) are typically placed in an antecedent-consequent format which resembles the following statement:

IF "some facts are true" **THEN** "establish some new facts"

where verity of the facts in the antecedent (the IF-part) effects the addition of the facts in the consequent (the THEN-part) to the knowledge base. The proposed planning scheme adopts the antecedent-consequent format (shown above) for all its rules.

Figure 3.11 illustrates the organization of Planning Level data bases and rule-bases. Rectangular boxes denote the different rule-bases while ovals represent the data-bases and fact groupings. Implicit elements are marked with slanted lines whereas explicit elements are denoted by a dotted pattern. Every fact holds information relating to the current status of the workstation or to the composition of the Master Plan. Consequently, two major divisions of facts are those facts which define the "state" of the workstation and those which define the "history" of the plan. "State" facts are always loaded into the Explicit Data Base (EDB) whereas "history" facts are held by the EDB and by the Implicit Data Base (IDB). Two groups of "state" (the terms

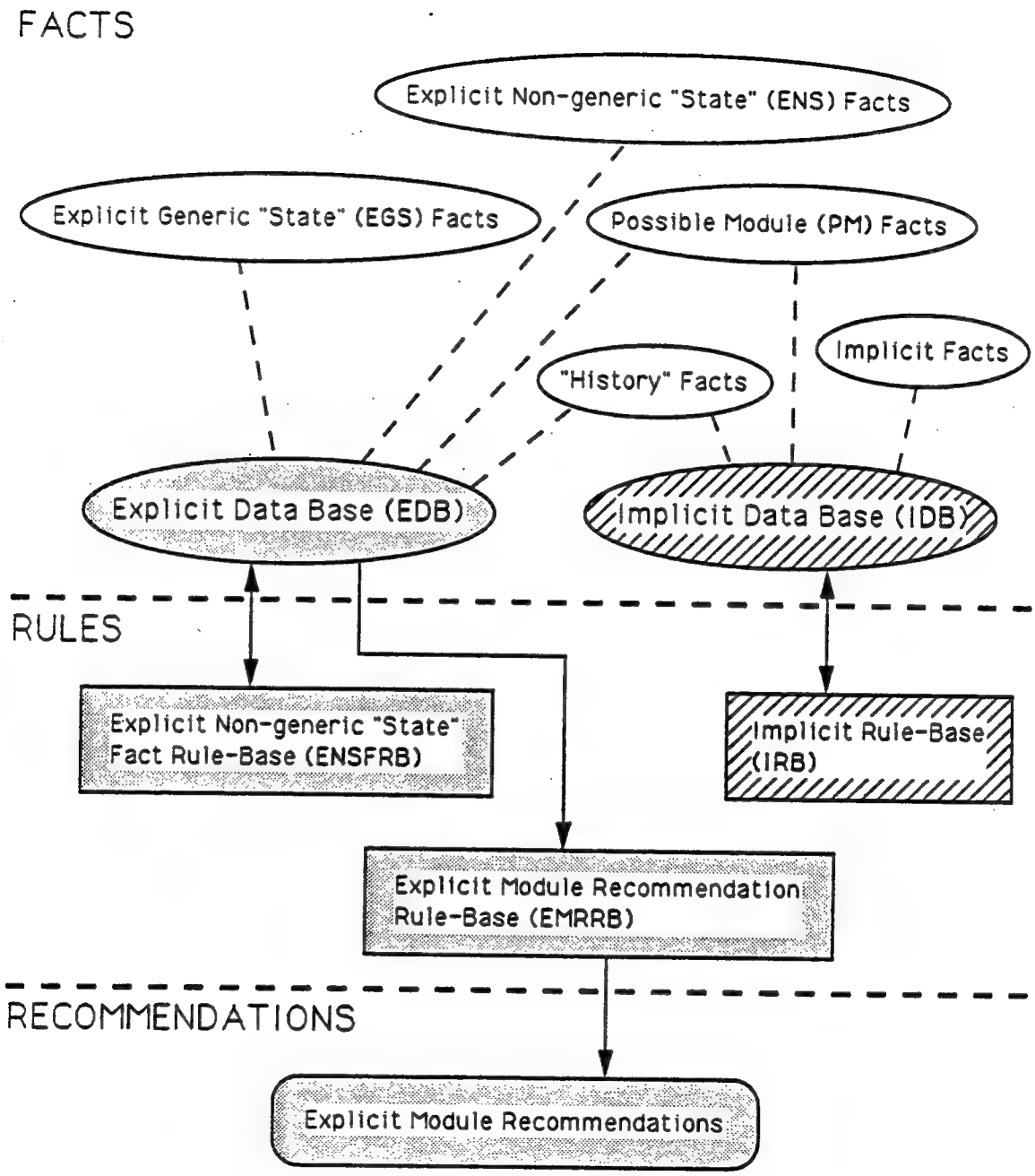


Figure 3.11. Organization of Data Bases and Rule-Bases

"state" and "history" will be placed in quotations when used in this context) facts are termed the Explicit Generic State (EGS) and the Explicit Non-generic "State" (ENS). Non-generic refers to facts unique to one particular configuration of workstation devices and objects while generic facts can apply to any configuration.

Additional types of facts include implicit and Possible Module (PM) facts. Implicit facts are generic and are used by the Implicit Rule-Base (IRB) to determine what implicit modules need be inserted into the plan. PM facts contain information regarding the suitability of applying a particular module to a Master Plan. These facts are consulted by the explicit and implicit rule-bases prior to recommending the inclusion of a module (the inclusion is automatic for the IRB) into a plan. Modules which are not supported by PM facts are guaranteed unsuitable for use in a Master Plan; the converse of this statement is, however, not always true.

The different types of facts are loaded into either the EDB or the IDB as shown by the dashed lines in Figure 3.11. The ERB and IRB systems remove "old" and add "new" facts to the data bases as needed. The terms "old" and "new" (henceforth placed in quotations if used in this context) refer to the current validity of a given fact. "Old" facts contain information which is no longer true (or cannot be supported), implying that they need removal from the data bases they occupy. "New" facts come about as the result of some

change in the condition of the workstation or the Master Plan and should be asserted to the appropriate data base.

Three rule-bases, two supplied by the EDB and one by the IDB, are the users of the various facts. The rule-base known as the Explicit Non-generic "State" Fact Rule-Base (ENSFRB) derives ENS facts based entirely on the facts currently held in the EDB. The other explicit rule-base, the Explicit Module Recommendation Rule-Base (EMRRB) takes facts from the EDB and yields explicit module recommendations. These recommendations are placed in the consequent portion of each EMRRB rule. The workstation operator selects and supervises the application of a particular module based on the EMRRB's recommendations. The EMRRB produces no additional facts for its supporting data base unlike the ENSFRB and the IRB. (This fact is indicated in Figure 3.11 by the arrowed line connecting the EDB to the EMRRB pointing in only one direction.) Specific details regarding the manner in which all these facts and rule-bases are implemented in the planning scheme are addressed in the following chapter.

Error Recovery

Errors which occur during the execution of a job plan by a workstation derive either from malfunctions in the hardware or from the fact that some anomaly has taken place which was not foreseen during planning. The control scheme provides the operator the capability to halt workstation operation by interrupting the software at the Coordination

Level (indicated by the line marked "Monitoring" in Figure 3.10 on page 55). The operator should observe workstation behavior during job plan execution and shut the system down if this behavior deviates from what is expected. Hardware failures do not necessarily imply that a job plan is incorrect. Once failed hardware is put back in working order, a given plan may prove to execute without error. Detection of hardware malfunctions is a formidable task and one that is not addressed in this research.

An error created as the result of improper planning should not be viewed as an error in the conventional sense but as a sign of incompleteness or poor reasoning in the job plan. The case of a robotic gripper transporting a part from one location to another is introduced to illustrate certain error recovery concepts. If the part should slip out of the gripper (this occurrence is not viewed as a hardware malfunction) during transport, the resulting condition may prove catastrophic because the robot, unaware of the part's absence, will attempt to place it at the designated location.

Automatic error recovery for this example requires that the workstation be able to detect the absence/presence of the part in the gripper. A device sensor must be installed for this purpose with supporting software written such that the information defining the gripper's condition can be accessed at a task-level. Inclusion of these components into the workstation hardware and software enables the

operator to insert a conditional module into the plan which would check to see if the gripper held the part before actually placing it. If the check proved negative, additional software could be added to the plan which would handle the error in an appropriate manner. Error recovery, for this example, involves extension of the job plan by means of the addition of new software (and hardware, if not already present). The plan, however, remains virtually unchanged except for the enhancement of its flexibility in dealing with a particular system error.

"Replanning" is defined as changing an existent plan in response to a new set of conditions and/or goals. The control scheme permits the replanning of old plans by the operator via the insertion of new modules into the module listing. Existent plan modules cannot be deleted or "resequenced" since these changes may affect the feasibility of the plan as a whole. The proposed strategy for handling system errors cannot implement continual, automatic monitoring of workstation hardware for deviate behavior at a task-level. Task modules are always executed in the sequence provided by the Master Plan. Workstation computers cannot perform one task simultaneously with an error monitoring task, which interrupts the system if an error is detected. On a process-level, however, an interrupt driven error detection strategy can be devised, but it must be incorporated entirely within the functions of a single module.

Chapter V provides an example which demonstrates the replanning approach discussed in this section.

Chapter Summary

A three-level scheme integrating Planning, Coordination, and Hardware Levels has been devised for controlling the operation of manufacturing workstations. The Planning Level combines the efforts of a workstation operator, expert programmer(s), and planning software (acting in conjunction with the operator) to plan and to program manufacturing activities which are, in turn, implemented by the Coordination and Hardware Levels. System representations are developed which distinguish the hardware and software issues associated with workstation control. One particular hardware configuration in which workstation devices and computing agents are distributed in a "star" arrangement is proposed as the prototype for the organization of the Coordination and Hardware Levels. Unique program structures are devised to direct workstation activities and are established to correspond with the job/task/process hierarchy into which all such activities are classified. These program structures, known as modules for task-level activities and functions for process-level ones, permit the separation of hardware-related information from procedural information and allow the differentiation of the data entering a structure (input) to the data leaving it (output).

The formation of the function sequences for each module is delegated to a system programmer who understands the process-level issues associated with each module and its related task. On a task-level, however, Planning Level software interacts with an operator to assemble the module sequence (plan) corresponding to the desired workstation job. This software operates in a semi-autonomous, off-line mode, providing the operator with feedback in the form of simulated workstation activity so that he/she can decide what tasks to incorporate into the job plan. Two rule-base systems are situated at the Planning Level, one of which (the ERB system) makes "knowledgeable" recommendations to the operator regarding what tasks to apply during a planning session. The other rule-base system (the IRB) acts autonomously inserting modules into a job plan needed to satisfy the data requirements of the recommended plan modules. The planning scheme ensures that a plan remains viable as new modules are added to it by coordinating the data which flows into and out of each plan module. The task-level plan is converted into process-level control programs which can be downloaded to the workstation computers for on-line operation.

CHAPTER IV

IMPLEMENTATION OF WORKSTATION CONTROL SCHEME

Introduction

The control and planning concepts and principles introduced in the previous chapter are developed into application software in this chapter. The implementation of the workstation control scheme requires extensive software which interacts with a human operator for determining the course of workstation activity. The various components of this software are presented with emphasis on the contribution of the rule-base systems and the operation of the planning algorithm. The means by which workstation features are characterized as informational which can be manipulated in software are discussed. Kinematic models of workstation devices and objects are developed to enable graphics simulation of the workstation as it performs operator-selected tasks.

Software Considerations

The hierarchical semi-autonomous control scheme software is composed of all the off-line elements within the Planning Level, as shown in Figure 3.10 (page 55). The software is developed to conform to PC-level computers similar to those proposed for workstation control in Chapter III. Consequently, planning a workstation job can be achieved using the same computers which execute the job plan

on-line using the workstation hardware, once planning is completed. Different computing equipment must be used for off-line planning when the workstation control computers must remain in an on-line mode.

Planning Level software is designed for use with two PC/AT computers connected by a RS-232 serial communication interface. One computer, labeled the Graphics Simulator (GS), performs the visual simulation of the workstation devices and objects as they carry out their planned activities. The other computer, termed the User Interface and Planner (UIP), conducts all the duties related to planning. Planning Level software is henceforth referred to as the Operator-Driven Controller And Planner (ODCAP) software package. ODCAP's name suggests the need for a human operator who can interactively assist the software in the management of workstation operations. The package is composed of programs written in Microsoft C and rule-bases developed using Arity Prolog as well as numerous text files containing workstation data.

All ODCAP software is contained in numerous data and program files stored in the permanent memories (hard disk drives) of the GS and the UIP. Understanding the organization and purpose of these files provides insight into how ODCAP works. Every workstation is composed of physical actuators, sensors, and objects; information which defines these items must be specified for utilization by the ODCAP software. Workstation hardware is classified to facilitate

the access of information by ODCAP during planning. Barbera, Fitzgerald, and Albus [60] recommend that a system should not be supplied with any information it cannot use. Simple kinematic models are required for devices and objects to minimize the data needed to define their geometry. The implementation of the Explicit and Implicit Rule-Base (ERB and IRB) systems by the ODCAP software is examined. The manner in which workstation information is represented and used by the rule-bases to aid the operator is discussed.

ODCAP File Organization

The inherent parallelism between a knowledge-based system and computer hardware is described by Valavanis and Saridis [61]. ODCAP requires two PC/ATs, the UIP and the GS, to store files and to execute programs. Both of these computers are controlled by the Disk Operating System (DOS) which enables file storage and program execution. DOS utilizes a special format for file labeling; all filenames are of the form "name.extension" where "name" refers to an arrangement of eight (or less) alphanumeric characters and "extension" signifies a similar arrangement of three (or less). A filename's extension is often used to group files by type; for instance, files with the extension ".EXT" contain executable program code.

DOS files are partitioned on their resident disk drives by means of directories and subdirectories. Labels for

directories and subdirectories start with the "\" to distinguish them from filenames. A file such as SETUP.EXE which resides in the \AAW directory and in the \EXE subdirectory would be given the single label \AAW\EXE\SETUP.EXE. All ODCAP files are placed in the \AAW directory of either the UIP or the GS hard disk drives. Figure 4.1 provides a comprehensive list of the program and data files (grouped by subdirectory) which constitute the ODCAP software package. This section covering ODCAP file organization concludes with descriptions of the various subdirectory file groupings and their contents. The purpose of individual files is discussed when relevant to a general understanding of how ODCAP works.

\AAW\PLAN Files

The subdirectory \PLAN in the UIP consists of files which are continually accessed and altered during planning. The four files listed with the ".PLN" extension are textual files (all data adheres to ASCII specifications) which hold information on the plan as it is created by ODCAP with operator assistance. SUB##.PLN contains the function listings (subplans) required for the n Controllers and the System Supervisor while MASTER##.PLN retains the single module listing known as the Master Plan. The introductory data section (defined in Chapter III) is contained in DATA##.PLN. MODS##.PLN contains information needed by ODCAP

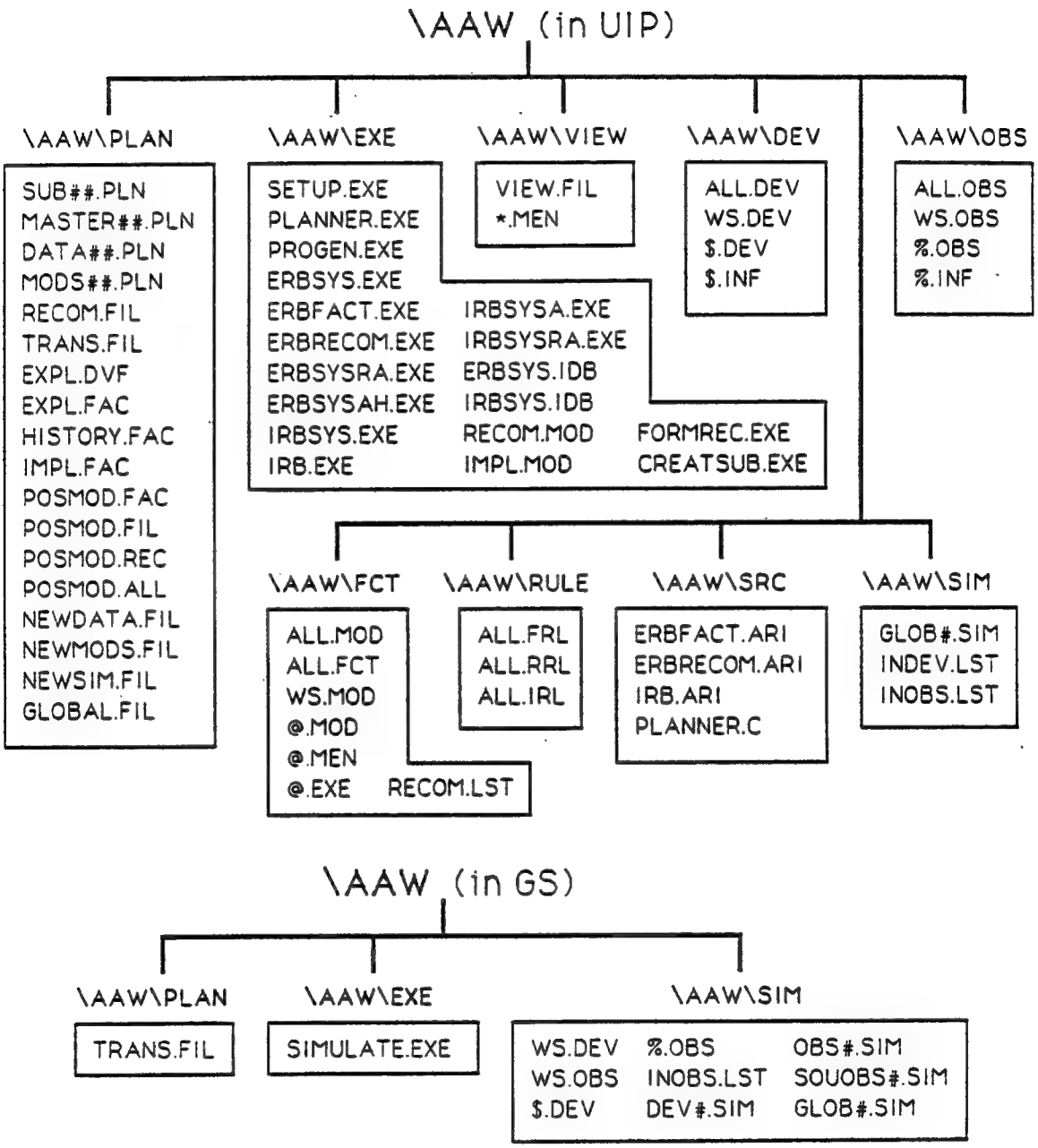


Figure 4.1. ODCAP File Organization

for the planning algorithm. The character set "##" symbolizes the plan identification number specified by the operator when the ODCAP software is executed; one hundred different plan values (from 00 to 99) are permissible.

The remaining files in \PLAN represent text files which act as information depositories during planning. Files in the subdirectory with the names "EXPL" contain the explicit facts prior to being included in the Explicit Data Base (EDB). HISTORY.FAC and IMPL.FAC contain the "history" and implicit facts respectively. Possible Module (PM) facts are kept in the files POSMOD.FAC and POSMOD.FIL. The list of the explicit module recommendations and the list identifying all the possible modules that the operator can apply to a given plan (a superset of the previous list) are held in POSMOD.REC and POSMOD.ALL respectively. The data file \AAW\PLAN\TRANS.FIL in the UIP and the GS contains kinematic information used for the workstation simulation.

\AAW\EXE Files

The \EXE subdirectories of the UIP and the GS list all (with the exception of the files marked @.EXE in \FCT) the executable programs in ODCAP. Four of these program files, SETUP.EXE, PLANNER.EXE, and PROGEN.EXE in the UIP and SIMULATE.EXE in the GS, are run by the operator; the remaining ".EXE" files are executed autonomously by ODCAP. SETUP.EXE is executed whenever the operator wishes to configure or reconfigure the workstation. The operator is

allowed to add or remove any device or object from the list of workstation resources. Off-line semi-autonomous planning begins when the UIP's PLANNER.EXE and the GS's SIMULATE.EXE are executed simultaneously. PROGEN.EXE carries out the automatic generation of language code programs from the Controller and System Supervisor subplans. The program files FORMREC.EXE and CREATSUB.EXE are executed autonomously within PLANNER.EXE.

The remaining files in \EXE in the UIP effect the operation of the rule-base systems. Filenames beginning with the letters "ERB" denote executable programs required for the Explicit Rule-Base (ERB) system while those starting with "IRB" represent Implicit Rule-Base (IRB) system files. The Explicit Data Base (EDB) and Implicit Data Base (IDB) are contained in the respective files, ERBSYS.IDB and IRBSYS.IDB. RECOM.MOD and IMPL.MOD are used by the Explicit Module Recommendation Rule-Base (EMRRB) and the IRB, respectively, as sources of information produced upon the activation of the rule-bases. The implementation of ODCAP's rule-base systems is addressed in greater detail in an upcoming section of this chapter.

\AAW\VIEW Files

The subdirectory \VIEW holds one file named VIEW.FIL which retains information on the operator's viewing perspective of the workstation simulation. ODCAP permits this perspective to be altered at the request of the operator.

The files in \VIEW with the extension ".MEN" contain user interface information required by the UIP for its screen displays. The character "*" symbolizes the name for each of these information files. Appendix C addresses the operation of the user interface and the graphics simulation.

\AAW\DEV and \AAW\OBS Files

Files in the \DEV and \OBS subdirectories of the UIP store information related to workstation resources. ALL.DEV and ALL.OBS contain lists of all workstation devices and objects respectively while WS.DEV and WS.OBS comprise smaller lists of those respective devices and objects which are selected by the operator to make up the current workstation configuration. The remaining files in these two subdirectories are the individual information files associated with each and every workstation device and object. The character "$" denotes a device name while the character "%" signifies an object name (device and object names are limited to eight alphanumeric characters). Kinematic modeling data associated with devices and objects is held in files with ".DEV" and ".OBS" extensions respectively. The remaining information required by ODCAP for the devices and objects is placed in the files with the extension ".INF".

\AAW\FCT Files

Information on the modules and functions which ODCAP uses to construct plans and subplans is kept in files under the \FCT subdirectory. Each module has a separate file

which comprises information such as the module's input and output data structures and its function listings. These files are represented by @.MOD where the character "@" denotes a shortened version (up to eight characters) of some module's name (conditional module sets are incorporated into a single ".MOD" file). ALL.MOD holds a list of all possible workstation modules while WS.MOD lists only those modules which can be utilized by the current hardware configuration. The file ALL.FCT contains all the needed information required for every function used in the modules listed in ALL.MOD. RECOM.LST includes explicit module recommendation information required by the EMRRB. This file is assembled automatically by ODCAP (as are all files with the name "WS") once the operator has selected the workstation devices and objects. The files named @.MEN and @.EXE consist of user interface data and simulation code respectively, required when an explicit module is chosen for inclusion into a plan.

\AAW\RULE and \AAW\SRC Files

All rules used by the two rule-base systems are stored in files under subdirectory \RULE in text format. Explicit and implicit rules are placed in the files ALL.FRL, ALL.RRL, and ALL.IRL respectively. The information contained in these rule files is needed when ODCAP assembles the Arity Prolog source code files in subdirectory \SRC. The files possessing an ".ARI" extension in \SRC correspond to the rule-bases and are assembled when the initialization program

SETUP.EXE is run. The compositions of the rule-bases vary with workstation configuration since rules, like modules, can be device- and/or object-specific. The Microsoft C source code file (PLANNER.C) for the planning system does not require reassembly when the workstation configuration is altered. The three ".ARI" files in \SRC are compiled into their executable counterparts (same name but different extension ".EXE") by SETUP.EXE; these executable files are then placed in subdirectory \EXE.

\AAW\SIM Files

Files associated with graphics simulation are listed under the \SIM subdirectories of the UIP and the GS. INOBS.LST and INDEV.LST are information depositories for the data defining the objects which enter the workstation environment and the initial device settings respectively. These files can be altered by the operator, as any other text file, before planning. GLOB#.SIM files (the character "#" denotes a single digit from one to nine) contain the parameter values required for workstation simulation and are created in the UIP and passed to the GS after a new module is added to the Master Plan. The addition of a plan module means that some task will be performed by the workstation; the graphics simulation of this task by the GS may require several (up to nine allowed) screen displays illustrating how the workstation appearance changes with the performance of the selected task. In addition to GLOB#.SIM files, the

GS requires the files: WS.DEV, WS.OBS, $.DEV, %.OBS, and INOBS.LST be passed to it (only once per planning session) from the UIP. The program SIMULATE.EXE in the GS needs the information contained in all these files to produce the graphics data stored in the files DEV#.SIM, OBS#.SIM, and SOUOBS#.OBS. Appendix C provides detailed information regarding the operation of the workstation graphics.

Resource Classification and Modeling

All physical workstation elements (termed resources) are segregated into three divisions: devices, objects, and computers. The division computers includes all computer related hardware, such as motor controller interface boards and communication apparatus, which receive software commands and data as input and convert them into electrical control signals. A device refers to any machine which alters or senses the workstation environment or assists other devices in performing these actions (such as a workbench). Devices generally are classified by their primary capability. The resources that devices act upon to create desired products are known as objects. Device and object resource divisions are further classified as indicated in the resource taxonomy shown in Figure 4.2. Divisions are separated into classes, classes into subclasses, and so on. Arrowed lines in the figure always proceed from a lower-level classification to a higher-level one. The classifications in Figure 4.2 (with

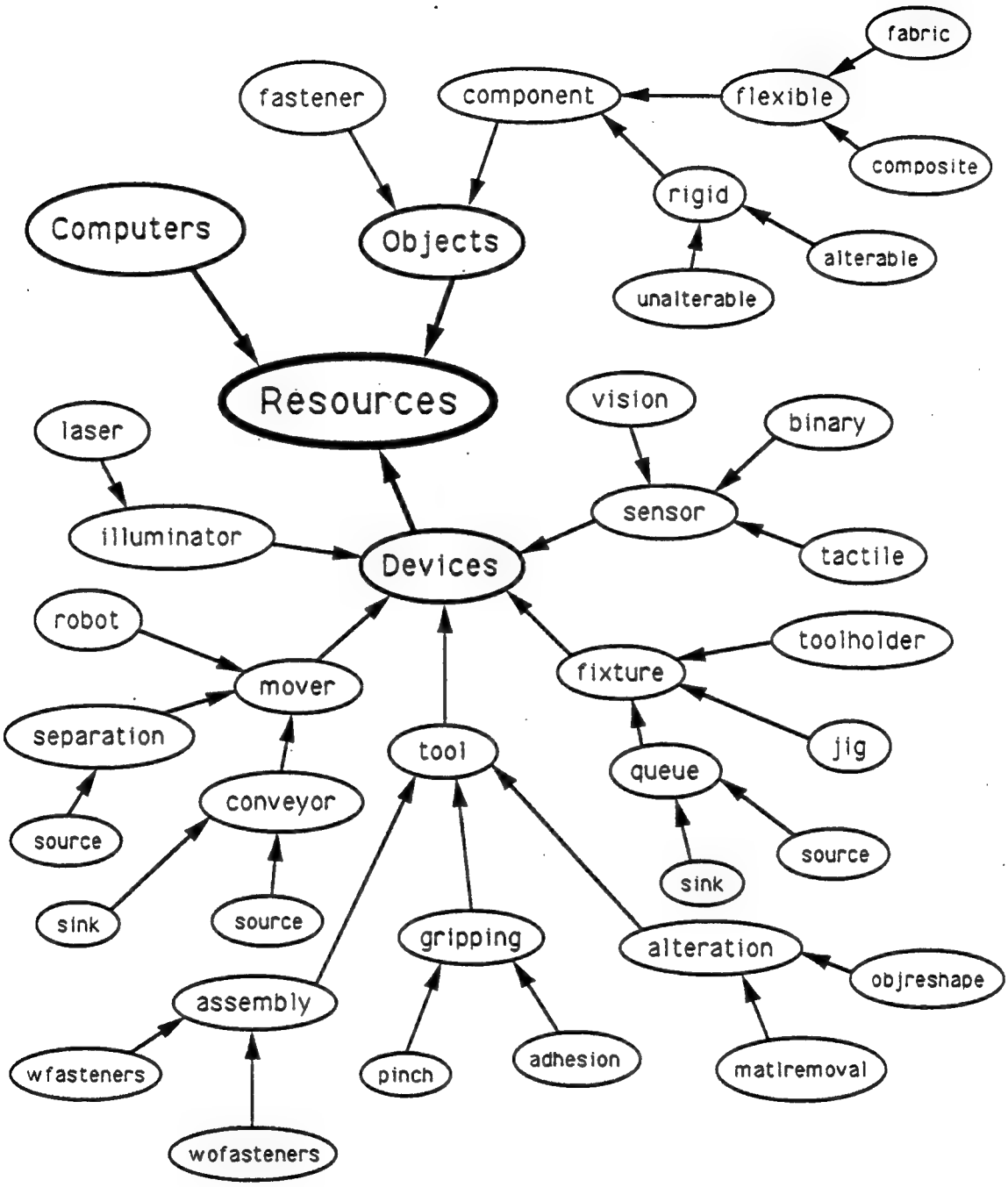


Figure 4.2. Resource Classification

the exception of divisions) should not be viewed as definitive; different classifications can be created dependent on the hardware configurations of the workstations involved.

Objects in this taxonomy are split into two classes, components and fasteners. Assemblies (not shown in Figure 4.2) are groups of components attached together with or without fasteners. Components are reduced into the subclasses of flexible and rigid to reflect the fact that the proposed control scheme is implemented on a apparel assembly workstation. The shirt collar introduced in Chapter I is identified as a flexible component.

The device resource division comprises five different classes, listed as follows: sensor, fixture, tool, mover, and illuminator. The class sensor includes any device which measures or detects change in some physical environment parameter. Cameras (vision subclass), break-beam detectors (binary subclass), and tactile arrays (tactile subclass) are examples of this class. Fixtures refer to any equipment which secures objects from moving as the objects are acted on by other devices (such as a worktable, subclass jig) or to any device which assists other devices (such as a toolrack, subclass toolholder). The primary feature of every fixture is that it contains no actuator mechanisms. Devices classified under the queue subclass are fixtures which store objects prior to their entrance into or exit from the workstation environment (subsubclass source and sink respectively).

The class tool is the broadest of the device classes, comprised of the three following subclasses: assembly, alteration, and gripping. Assembly tools are utilized to join object components together either with fasteners (such as a wrench, subsubclass wfasteners) or without fasteners (such as a welder, subsubclass wofasteners). Alteration tools transform objects either by removing material from them (such as a drill, subsubclass matlremoval) or by reshaping them (such as a metal bender, subsubclass objreshape). Changes in an objects attachment from one device to another are effected by gripping tools. For example, a robot gripper (subsubclass pinch) acquires an object from a table by changing the object's attachment from the table to itself.

Mover is a term given to the class of devices which can alter the position and orientation (the pose) of objects or other devices in the workstation environment. Examples of movers include robot manipulators (subclass robot) and conveyor belts (subclass conveyor). The class illuminator consists of devices which affect the lighting of the workstation environment (such as a laser, subclass laser) and are often used in conjunction with vision sensors to determine object features.

The classifications presented in the resource taxonomy of Figure 4.2 provide a means for distinguishing devices and objects based on their utility. Numerical models defining the kinematic aspects of devices and objects are needed in order to properly plan the actions performed by and on these

resources. Lam, Pollard, and Desai [53] refer to such models as topological models. Topological relationships between objects in assemblies are modeled using tree-like representations by Monckton and Toogood [62]. Jain and Donath [63] employ homogeneous transformations (as do other researchers) to numerically define the poses of system devices and objects. Knowing the geometry of devices and objects facilitates the specification of sensing tasks involving those devices and objects [64]. The topological models developed for this research account for device and object kinematics only. Dynamic force considerations such as those modeled by Caudill et al. [24] are omitted from the developed models due to their complexity. The use of matrices for representing pose transformations for devices and objects as described by Paul [65] is adopted for this research.

The pose of each workstation device or object is determined by a transformation between a single coordinate frame (called the base link frame) attached to the device or object and a common reference frame. The single workstation reference frame is termed **BASE** (all coordinate frame names consist of boldfaced, capitalized alphanumeric characters). A transformation between two frames is characterized by a 4x4 matrix which defines the orientation and position of one frame relative to the other. Figure 4.3 illustrates the relationship between a device base link frame and an object base link frame and the stationary frame **BASE**.

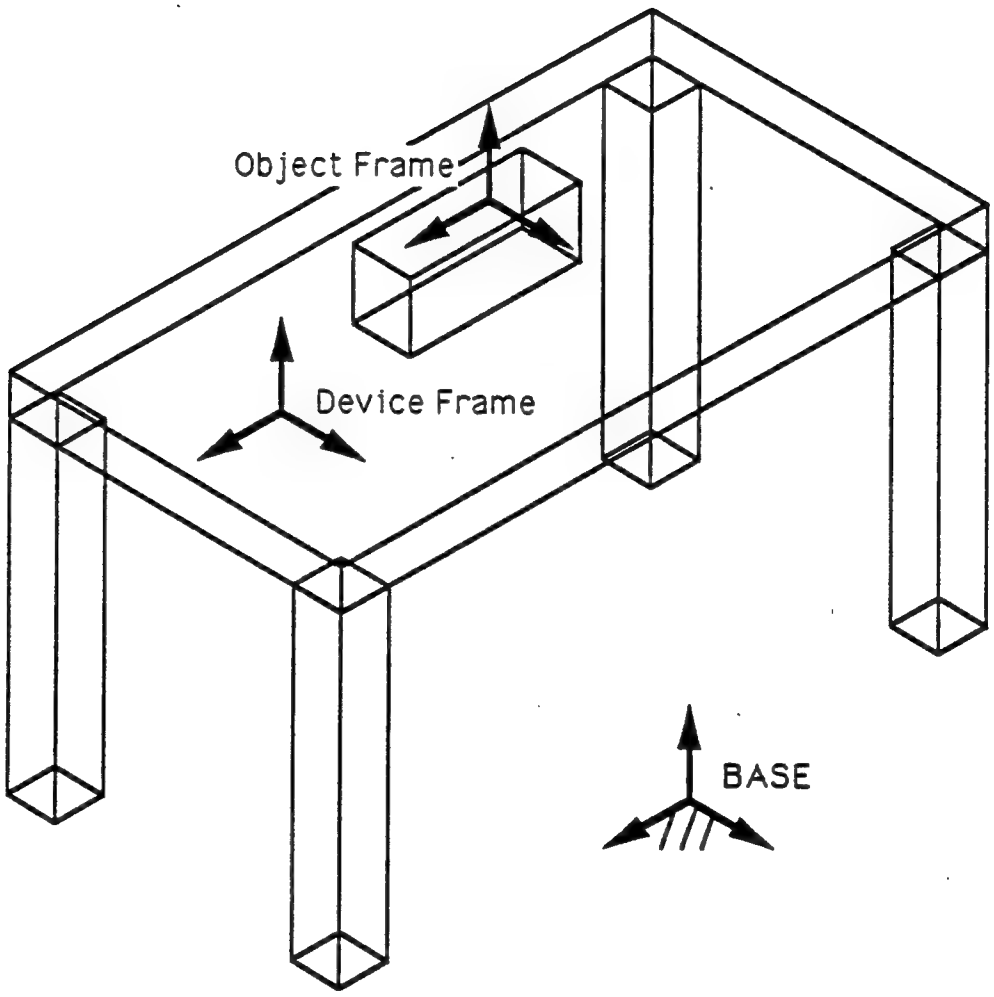


Figure 4.3. Device and Object Frames

Devices and objects are modeled as a wire frame composed of points and lines; the placement of these points and lines within each device and object model will determine the appearance of these resources in simulation. Model points are always referenced to some frame attached to the device or object and are numerically defined with a four-row vector (see Paul [65]). Lines are created when two points are connected by a line segment. Realistic geometric models are produced when many different points and lines are specified for each device or object. Figure 4.4 provides an example of a wire-frame model for a block-like object comprising eight points (labeled 1 through 8) and twelve lines (indicated with circled numbers). Objects possess a single base link frame with any number of lines and points (dependent on the amount of detail desired for the simulation) referenced to it. Device models include one base link frame plus any number of additional coordinate frames, each of which can have points and lines referenced to it. Device frames are related to one another using one of three transformations, defined as: fixed, prismatic, and revolute. All frames in a device model must be connected to the base link frame either directly with a single transformation or indirectly through another frame.

A gripper device is used to illustrate device model kinematics. Two views of the gripper are included in Figure 4.5, one with the jaws of the gripper in an open position and one with the jaws closed. The gripper's base link

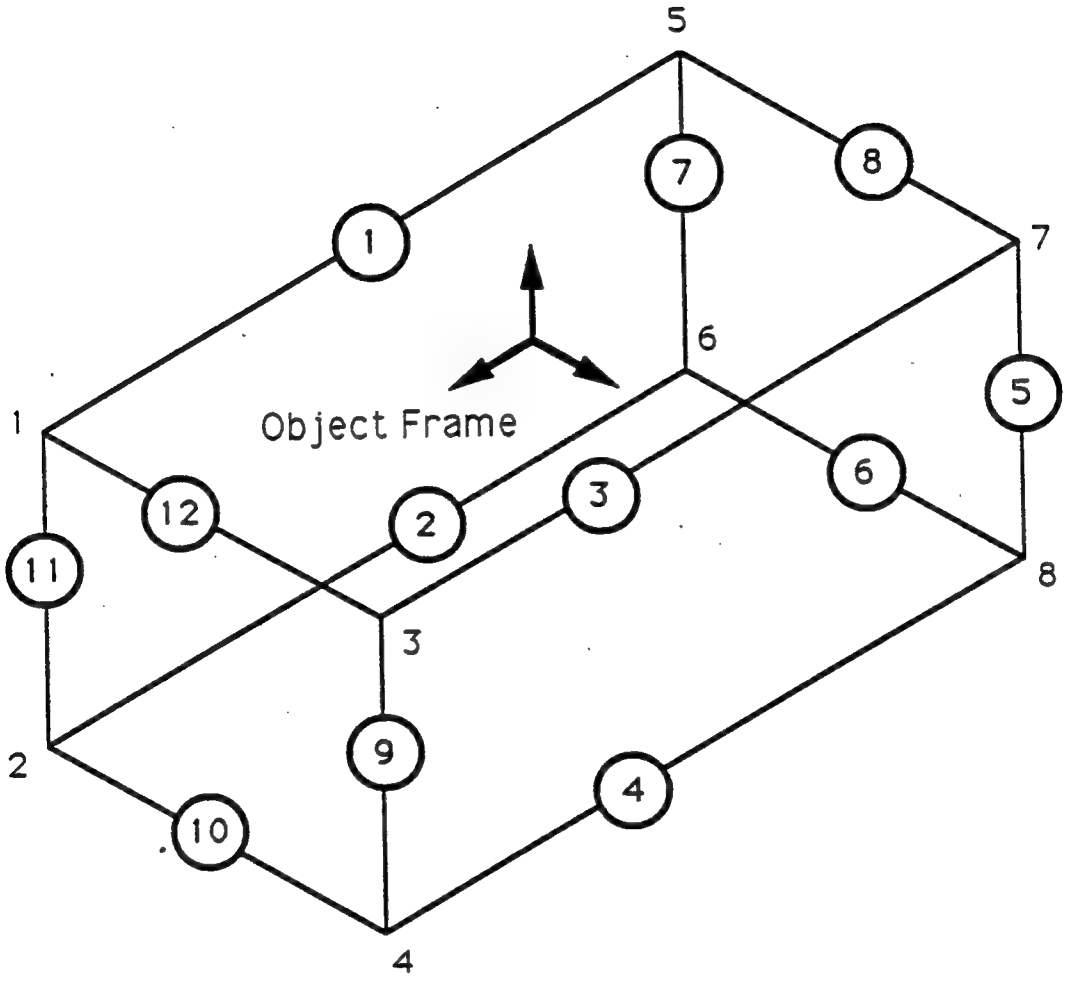


Figure 4.4. Object Model

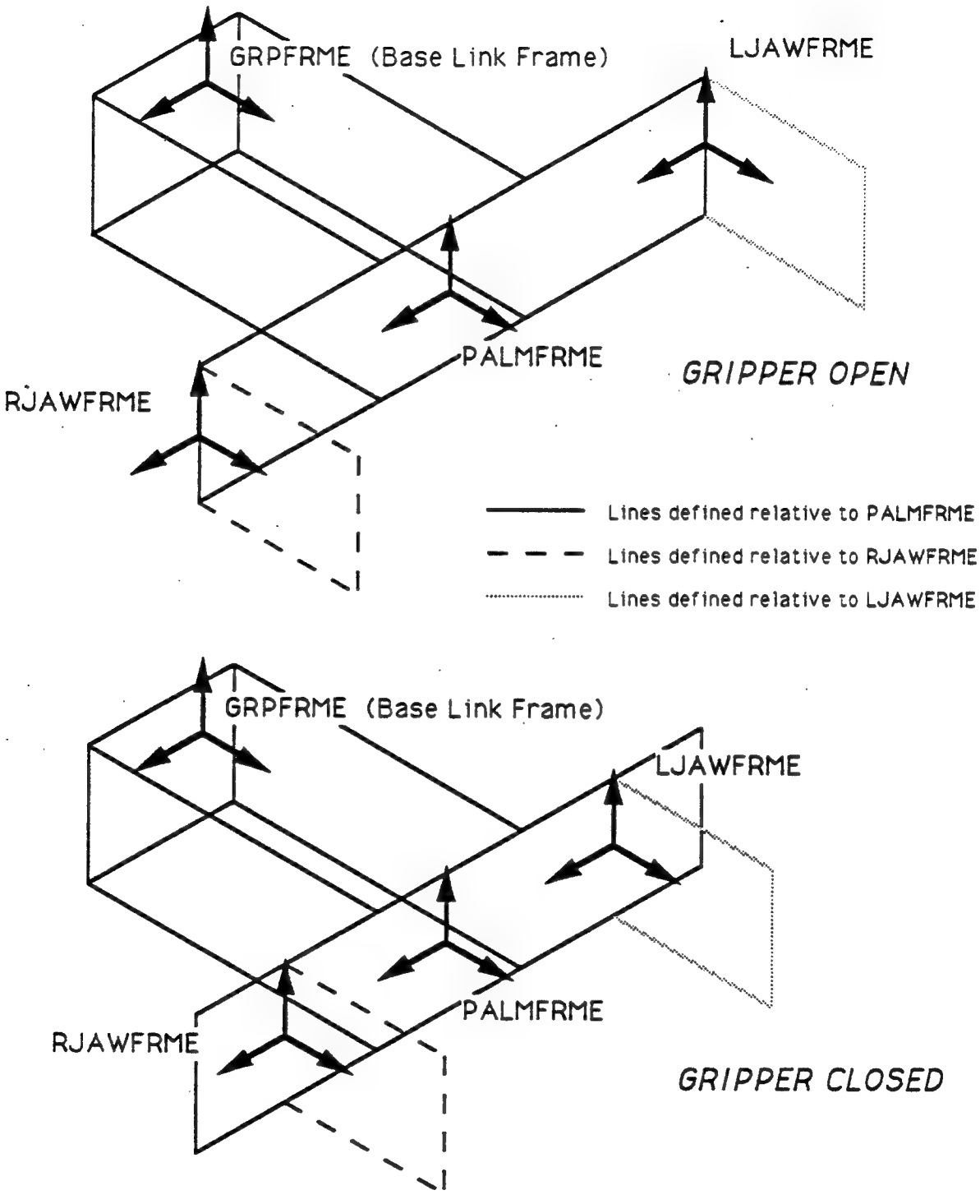


Figure 4.5. Device Model of Gripper

frame is labeled **GRPFRME**; the lines referenced to **GRPFRME** are indicated in the figure along with those lines connected to the other frames in the gripper model. The frame **PALMFRME** is linked to **GRPFRME** with a fixed transformation, implying that the geometric relationship between these frames can never change. **LJAWFRME** and **RJAWFRME** are individually connected to **PALMFRME** using prismatic transformations. A prismatic transformation between two frames allows one frame to vary its pose in one linear direction (along a single axis) with respect to the other frame. For the gripper model, the prismatic transformations between **LJAWFRME** and **PALMFRME** and between **RJAWFRME** and **PALMFRME** enable the jaws of the gripper to open and close. Revolute transformations cause frames to vary their poses in a single rotational direction (about one axis). A single device variable is required for every revolute and prismatic transformation. Denavit-Hartenberg matrices are utilized to specify revolute and prismatic transformations in device models [65]. This method was selected over comparable methods because of its near universal adoption by robotic researchers.

Two device variables must be specified for the gripper, just described, to fix uniquely the position of its jaws. These variables are included into a data set which comprises information associated with all the changeable aspects of workstation hardware. The variables in this data set are known as global parameters and incorporate the variables related to prismatic and revolute transformations. Often

devices have actuators which can attain a limited number of positions; such actuators are termed "switches". Each switch has a finite number of states associated with it. For instance, a pneumatic cylinder labeled GRPSTATE, which serves as the mechanical actuator for a gripper, can be considered a switch with the states of OPEN and CLOSED. Switch states and labels are specified as FINITE-STATEs when comprising function input data (see Chapter III). Workstation hardware can be altered by the setting of switches to different states; consequently, switch states are included in the global parameter set.

In addition to device switches and variables, the transformation between a device's or object's base link frame and **BASE** must be set numerically to establish workstation geometry. All object fames are linked to device frames whose positions and orientations are known. If the location of a device is established, the poses of those objects attached to it are also fixed. The matrix transformation between every object base link frame and some device frame is held as a global parameter (along with the name of the device frame) since these transformations determine the pose of workstation objects.

Device base link frames are linked to **BASE** either directly with a fixed transformation or indirectly through other device frames. A fixed transformation is not considered as part of the global parameters because a device possessing such a transformation is deemed "immovable" (the

converse of objects which can be relocated anywhere in the workstation). The indirect method of linking device frames permits some devices to be "movable" by having changeable attachments to other devices (similar to objects). An example of a movable device is a tool which can be attached to a robot wrist and detached when not needed. A movable device model requires the inclusion of the device name (and device frame) to which the movable device is attached as well as the matrix transformation, which defines the geometry of the attachment, into the set of global parameters.

Assemblies are defined as collections of objects which are interconnected. The specification of an assembly for the global parameter set only requires a list of those objects which constitute the assembly. All objects within an assembly must be attached to a single, common device frame. The global parameters when combined with the static kinematic data (held in the ODCAP files \AAW\DEV\$.DEV and \AAW\OBS\%.OBS) comprise all the information needed to establish workstation kinematics. Global parameter values are altered whenever action modules are added to a job plan. The execution of an action task involving a robot, for example, will affect the device variables which determine the robot's arm geometry. Static kinematic data can only be changed when a workstation is configured (or reconfigured) before planning begins.

Explicit and Implicit Rule-Base Systems

ODCAP's rule-base systems assist the planning system and the workstation operator in deciding which modules need be appended to the plan to achieve the desired job. The configuration and purpose of the two explicit rule-bases (the ENSFRB and the EMRRB) and the single implicit rule-base (the IRB) have been discussed in Chapter III. The Arity Prolog code for the three type of rules (one for each rule-base) is listed in the files ALL.FRL, ALL.RRL, and ALL.IRL (in the \RULE subdirectory) under the labels F###, R###, and I### respectively where the "###" denotes a three-digit identification number. Attached to each rule is a list of device and object resources that the rule requires in order for it to be applicable to a job plan. The initialization program SETUP.EXE refers to these resource lists when deciding which rules should be included in the source code files ERBFACT.ARI, ERBRECOM.ARI, and IRB.ARI (in \SRC).

Prolog is a language intended for AI applications, operating in a different manner than conventional languages such as "C". Conventional languages tend to be procedural whereas Prolog programs represent a description of the encoded system rather than a sequence of prescribed system actions. Rules composed using the antecedent-consequent format can easily be transcribed into Prolog code. The rule-bases written in Prolog are merely lists of such rules. When the rule-base programs are executed, data base facts are applied to the antecedent portion of the rule. If all

the conditions specified in the antecedent can be confirmed by data base information, the facts contained in the rule's consequent are applied to the data base. The rule is said to have been "fired" when this result occurs. The execution of rule-base programs terminates when the data base is exhausted and all the rules which can be fired have been fired.

Facts for the proposed control scheme are implemented using first-order predicate calculus. Fikes and Kehler [66] identify this method as one means for storing knowledge in a knowledge base. The representation of a fact with first-order predicate calculus is demonstrated by example. The fact, with syntax defined as follows:

person(john, plumber)

contains information about some "person", namely that his name is "john" and his occupation is a "plumber". The item outside the parentheses (in this case "person") signifies some general category for which items inside the parentheses ("john" and "plumber") provide specific information. The items in a fact's property list ("john" and "plumber") can be changed when new facts are needed relating to the same general category ("person"). The facts:

person(bonnie, teacher) and

person(don, engineer)

are furnished as examples of facts with different properties. The predicate calculus forms for every fact type used

in this research (except for Explicit Non-generic "State" (ENS) facts) are presented in Appendix D.

The rules in the Explicit Non-generic "State" Fact Rule-Base (ENSFRB) examine all the Explicit Generic "State" (EGS) facts in the EDB and assert new ENS facts when possible. Examples of ENSFRB rules are provided (along with EMRRB and IRB rules) in Appendix D. EGS facts are based solely on global parameters, providing a qualitative description of workstation status while global parameters (discussed in the previous section) define this status quantitatively. EMRRB rules can have EGS, ENS, PM, and "history" facts incorporated in their antecedents. The consequent portion of EMRRB rules, however, consists of a single explicit module recommendation. Each recommendation includes the name of the explicit module, that ODCAP advocates for inclusion in the plan, as well as specific values for every FINITE-STATE in the module's input. The ENSFRB and the EMRRB, when activated, can yield any number of new ENS facts and explicit module recommendations respectively, dependent on the number of rules fired. The operation of the IRB, on the other hand, is suspended after only one rule is fired.

Possible Module (PM) facts are based on the FINITE-STATE values that each module can attain as input. Included in every module's data file (@.MOD) are PM requirements which determine what devices and objects can be selected as FINITE-STATEs. A given module, for example, might require

that one of its FINITE-STATEs represent some device of a certain resource class and/or subclass. A device with these particular resource classifications must be present in the workstation before this module can be considered for use in a job plan. PM facts specify the names of every module whose FINITE-STATEs satisfy PM requirements. These facts also list all the FINITE-STATE values possible for each module. "History" facts contain information regarding the status of a job plan. Such information as the names of the variable-structures which could provide the input for new modules is represented within "history" facts. Examples of PM and "history" facts can be found in Appendix D.

The Implicit Rule-Base is used to satisfy the input variable-structure data requirements of modules chosen by the workstation operator for inclusion in a job plan. The operation of the IRB is explained with a planning example. The example involves the module **pick_place()()** with FINITE-STATEs and variable-structures defined as follows.

```
pick_place
(ROBOT, BLOCK, traj:ROBOT:SS, prop:BLOCK:SS)
()
```

The module has two FINITE-STATEs, the first one representing a device and the second an object. The values selected for the FINITE-STATEs in the example are ROBOT and BLOCK. The first input variable-structure, named "traj", contains information needed by the device ROBOT to carry out the task corresponding to **pick_place()()**. The other input variable-structure labeled "prop" represents data associated with the

object **BLOCK** which is also required by the task. Both input variable-structures are held in the workstation computer known as "SS". No output variable-structures are specified for **pick_place()()**.

If the example module, with the FINITE-STATE values shown above, is to be incorporated into the Master Plan, the module's input variable-structures must be part of some previous module's (or modules') output. For this example, it is supposed that neither of the two variable-structures for the new module can be found in the output of modules in the existent plan. The purpose of the IRB is to recognize this fact and to obtain implicit modules which can be inserted before the new explicit module to satisfy its input data requirements. The IRB might propose the introduction of the implicit module **intro_traj()()** such that the Master Plan resembles the following.

```
{Existent Master Plan situated above the
 location of the newly inserted explicit module}

 intro_traj
 (ROBOT)
 (traj:ROBOT:SS)
 pick_place
 (ROBOT, BLOCK, traj:ROBOT:SS, prop:BLOCK:SS)
 ()

{Existent Master Plan situated below the
 location of the newly inserted explicit module}
```

The execution of the IRB ceases after proposing the **intro_traj()()** module; but, as can be seen in the module listing, the input data requirements of **pick_place()()** have still not been met. The IRB is activated repeatedly until

these requirements are assured or until the Implicit Data Base is exhausted and no more implicit rules can be fired. For the example, the IRB's second activation produces the implicit module **intro_prop()()** such that the Master Plan now resembles the following.

```
{Existent Master Plan situated above the
 location of the newly inserted explicit module}

 intro_prop
 (BLOCK)
 (prop:BLOCK:SS)
 intro_traj
 (ROBOT)
 (traj:ROBOT:SS)
 pick_place
 (ROBOT, BLOCK, traj:ROBOT:SS, prop:BLOCK:SS)
 ()

{Existent Master Plan situated below the
 location of the newly inserted explicit module}
```

The IRB does not need to be activated a third time, for the input data requirements of **pick_place()()** are satisfied with the addition of the second implicit module.

In addition to proposing implicit modules which are placed directly above the newly inserted explicit module, the IRB can produce implicit modules which are intended for placement at the top of the Master Plan. This type of implicit module is responsible for introducing data which should only be input once. If the IRB fails to find appropriate implicit modules which allow the insertion of a proposed explicit module, the explicit module is considered (currently) inadmissible into the plan, implying that a new module must be selected by the operator.

Planning Algorithm

The "main engine" of Planning Level software is the planning algorithm which utilizes the user interface, the graphics simulation, and the rule-base systems previously described. The planning algorithm creates each job plan one module or module set at a time with the workstation operator selecting the tasks to which the modules correspond. Graphics Simulation (see Appendix C) of the workstation is conducted simultaneous with the execution of the planning actions. This feature allows the operator to monitor the workstation status and to view the accumulative effects of the planned tasks on the workstation environment. The operator can choose any module which ODCAP deems feasible, but he/she is provided with a list of recommendations derived from rule-base analyses of the "state" of the workstation appartus and the "history" of the current plan. Each completed cycle of the planning algorithm oversees the addition of one new explicit module and any number of implicit modules to the Master Plan. The job plan and the planning session can be concluded whenever the operator deems a successfully applied module to be the final module included in the Master Plan.

Four modes of operation are possible for the planning algorithm, namely regular planning, replanning, simulation, and re-simulation. The first mode entails the creation of a new job plan by the planning algorithm and by the workstation operator. The other three modes require an existent

plan which must be created during a regular planning session. The "replanning" mode operates similarly to the regular planning mode except that the operator is permitted to add new modules to the existent plan. The simulation mode should be selected by the operator if he/she wishes to see the performance (in simulation) of all the tasks corresponding to an existent plan. The "re-simulation" mode is utilized to reconfigure an existent plan's introductory data section if workstation parameters have been altered following the creation of the plan. When placed in a re-simulation mode, ODCAP determines which tasks should be performed based on the old module listing; however, the operator must supply new information regarding how these tasks are to be performed.

The program code for the planning algorithm is long and complex consisting of numerous procedures and intricate branching between the procedures. A single flowchart is used to illustrate the algorithm with all its procedures grouped into eleven separate divisions known as "code blocks". Each code block represents a collection of program elements which are related by a common purpose. Figure 4.6 depicts the flowchart and shows how branching can occur between code blocks. Every code block is labeled with a single title (with every letter capitalized and boldfaced); the code block titles are enclosed by boxes which have rounded corners or squared ones, indicating blocks which involve or do not require operator interaction respectively.

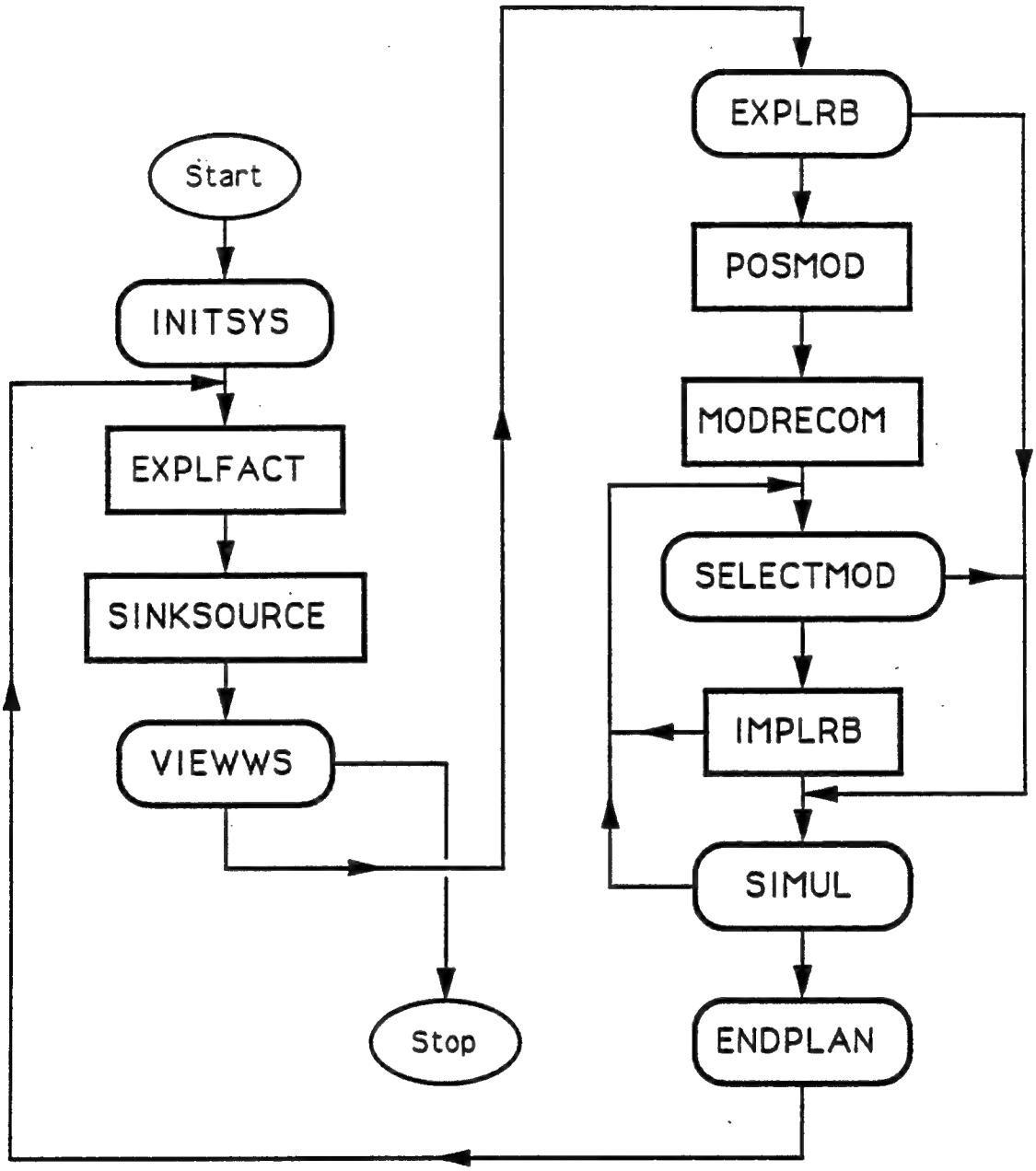


Figure 4.6. Planning Algorithm Flowchart

Branching can take place from any block to any other block dependent on the results of the actions performed within the block. This section examines each code block, explaining briefly their purpose and operation. Appendix E provides more complete programming details including "pseudocode" listings for each code block.

The first code block of the planning algorithm **INITSYS** obtains the planning mode and the plan identification number from the workstation operator and assigns values to every global parameter related to a device property. The transfer of files containing static simulation data (from the UIP to the GS) occurs in this block since these operations need only be performed once. The implicit modules situated at the top of the existent plan (the plan files are indicated by the plan identification number) are executed when the algorithm is in a mode other than regular planning. Execution of the procedures in **INITSYS** is never repeated for a single run of the algorithm. The block **EXPLFACT** oversees the generation of EGS facts based on global parameter values and the subsequent loading of these facts into the EDB.

SINKSOURCE conducts the "sinking" and "sourcing" of objects into the workstation environment. Sinking and sourcing refer to the respective removal of finished products from and the entrance of objects into the workstation. Explanations pertaining to the theory and implementation of these two features are contained in Appendix F. Some EGS facts may be affected by sinking and sourcing and must be

reloaded into the EDB at the end of **SINKSOURCE**. The operator is permitted to alter his/her viewing perspective (see Appendix C) of the workstation simulation in the code block **VIEWWS**. The simulated execution of a newly planned task occurs following this alteration.

EXPLRB autonomously activates the ENSFRB for the generation of ENS facts. If the operator is conducting the "replanning" of an existent plan, ODCAP will query him/her on his/her intention regarding the insertion of a new module into the plan. If a module insertion is desired by the operator, the algorithm will continue with **POSMOD**; otherwise, the algorithm branches to **SIMUL**. Automatic branching to **SIMUL** can occur for regular planning when looping by conditional modules (see Appendix A) has forced the repeated execution of some previously planned module. The simulation and re-simulation modes always undergo automatic branching to **SIMUL** at the conclusion of **EXPLRB**. PM facts are generated and loaded into the EDB by the procedures in **POSMOD**.

MODRECOM directs the activation of the EMRRB which produces explicit module recommendations with particular values chosen for each module's set of FINITE-STATEs. The creation of a list (in file \AAW\PLAN\POSMOD.ALL) containing all the feasible modules which can be deduced from the PM facts is performed by this code block. **SELECTMOD** oversees the presentation of the explicit module recommendations to the workstation operator. For each module, ODCAP explains the task which will be performed if the module is selected for

inclusion into the plan. The module which is chosen is termed the Selected Explicit Module (SEM), indicating not only the module's name but a particular set of FINITE-STATE values designated by ODCAP. The operator is allowed to pick the SEM from the list of possible modules if he/she is not satisfied with any recommended module (or if no modules can be recommended). Automatic branching to **SIMUL** occurs only if the input data requirements of the SEM are satisfied without the addition of implicit modules to the plan.

The IRB is activated as often as necessary in block **IMPLRB** to ascertain implicit modules whose output data structures match the SEM's input ones. Failure of the IRB in this goal forces branching back to **SELECTMOD** where the operator must choose a new SEM. The simulation of the tasks corresponding to the SEM and its inherited implicit modules is achieved in **SIMUL**. The global parameters which are affected by the simulated tasks are altered in value. The SEM may fail to execute properly (in simulation) if the status of the workstation hardware does not support its success. If the SEM's task, for example, concerns the acquisition of some object by a robot and the object is located outside of the robot's workspace, the acquisition task is not possible. Only by means of simulation can such failures be determined since all the other planning elements in ODCAP deal with qualitative not quantitative information. Failure of the SEM's task would lead to the automatic branching of the algorithm back to **SELECTMOD**. **SIMUL** is considered an

operator interactive code block because the addition of new modules may require him/her to provide information concerning how the SEM's task is accomplished. "History" facts are generated as the last operation in **SIMUL**.

The final code block **ENDPLAN** queries the operator on the termination of the planning session (resulting in a Master Plan composed of all the modules selected during the session) when the algorithm is in a planning or replanning mode. **ENDPLAN** procedures can generate process-level subplans from the Master Plan if the operator desires. The operator can designate that the session be terminated in **ENDPLAN**, but the algorithm will conduct one final task simulation in **VIEWWS** before actually stopping. For the simulation and re-simulation modes, the planning algorithm is terminated when ODCAP determines that the last module in the existent plan has undergone simulation.

CHAPTER V

JOB PLANNING DEMONSTRATION

Introduction

The planning scheme implemented by the ODCAP software is applied to a hypothetical assembly problem. The demonstration involves a workstation consisting of a robot and eleven other devices which carry out the manufacture of an assembly from four objects. Specific details are provided on every workstation resource and the tasks required to achieve the assembly job. The initialization procedures that the workstation operator must perform, prior to job planning with ODCAP, are discussed. Appendix G should be consulted for exact details regarding the operation of ODCAP.

Job planning is conducted in a step-by-step manner with each new step effecting the addition of a new task to the task sequence and the task's corresponding module (or module set) to the Master Plan. Process-level subplans constituted from the modules' composite functions can be assembled automatically by ODCAP at the conclusion of the planning session. The creation of the plan corresponding to the assembly problem is examined in depth for the first few repetitions of the planning cycle. The interaction between the operator and ODCAP during each cycle is presented. Complete Master Plan and partial subplan listings are

included to illustrate the finished products created by an ODCAP planning session. The chapter concludes with a "replanning" example involving extension of the job plan.

Assembly Example

The example problem concerns the assembly of a product known as a WIDGET from two object components and two object fasteners. The assembly job involves robotic manipulation of these objects and machine vision sensing of the components. Job tasks and workstation hardware are kept intentionally simple since the purpose of the example is to demonstrate ODCAP job planning not automated manufacturing. Workstation control derives from three computers possessing the following labels: the System Supervisor (SS), the Vision Controller (VC), and the Robot Controller (RC). The VC and the RC direct the image processing of the workstation's one vision sensor and the movement of the robot arm respectively. The determination of robot motion trajectories is based on the visual analyses of object components and is achieved within the SS. This central computer also controls all device actuators except those of the robot. Task and device and object descriptions are provided in this section with devices and objects labeled in the form of FINITE-STATEs.

The WIDGET assembly is constructed from two object components named BOTCOMP and TOPCOMP which are brought into the workstation (see Appendix F for information on object

sourcing) by a conveyor belt termed INCONV. A single workstation vision sensor (labeled CAMERA) is situated above INCONV to determine the position and orientation of incoming BOTCOMP and TOPCOMP objects. The components are placed on INCONV in groups of two (one BOTCOMP and one TOPCOMP) with groups spaced at roughly equivalent intervals. The possibility exists that a given group may comprise only one component or two of the same components or three components. When any of these conditions occur, the component(s) in the improper group should be discarded to a fixture device known as REJBIN. If a group contains one BOTCOMP and one TOPCOMP, the objects are subsequently transported by the robot to an assembly worktable labeled TABLE.

The robot device (named ROBOT) about which the example workstation is centered is a four-degree-of-freedom SCARA manipulator with a tool gripping device connected to its wrist. This attached device (called TOOLGRP) enables ROBOT to exchange end-effector tools for different tasks. Two such tools are included in the example, one named GRIPPER and the other BOLTDRV. GRIPPER is utilized to acquire and transport BOTCOMP and TOPCOMP as well as finished WIDGET assemblies; the device has a two-position (OPEN and CLOSE) actuator switch (labeled GRPSTATE) whose setting indicates the status of GRIPPER's jaw opening. The other "movable" tool called BOLTDRV obtains object fasteners and uses them to join TOPCOMP to BOTCOMP. Both GRIPPER and BOLTDRV are initially placed on a toolholder device labeled TOOLRACK.

The robot maneuvers the attached device TOOLGRP such that it connects with either GRIPPER or BOLTDRV while they are held by TOOLRACK. A two-position (ON and OFF) switch mechanism (labeled TGRPSTAT) locks or unlocks one of the tools to TOOLGRP.

Following the placement of one TOPCOMP and one BOTCOMP onto TABLE, WIDGET manufacture proceeds with the mating of TOPCOMP to BOTCOMP by means of the robot and GRIPPER. The tool GRIPPER is put back onto TOOLRACK and exchanged for BOLTDRV so that object fasteners can be retrieved. WIDGETs are assembled with an object fastener termed BOLT; two BOLTs are required for every WIDGET. A fixture device labeled BOLTFDR acts as the source for incoming BOLTs. The tool BOLTDRV can acquire one BOLT from BOLTDRV in the same manner that GRIPPER retrieves a component from INCONV. The robot uses BOLTDRV to insert and attach two BOLT fasteners to TOPCOMP and BOTCOMP to produce a finished WIDGET. The tool BOLTDRV is subsequently returned to TOOLRACK and exchanged with GRIPPER so that the robot can transport the WIDGET assembly from TABLE to an output conveyor device named OUTCONV.

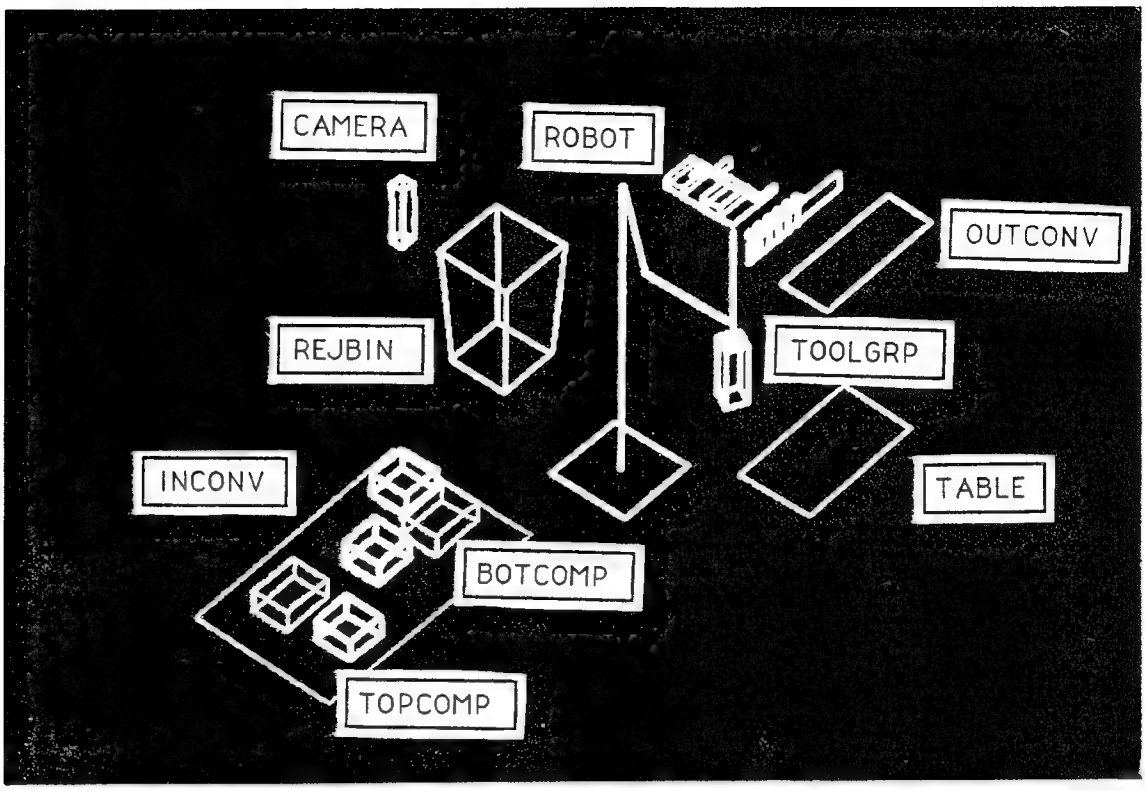
The assembly job should attempt (improper component groups on INCONV may prevent successful WIDGET manufacture) to make three WIDGETs before ceasing workstation activity. The belt on the conveyor device INCONV must be advanced by an amount equal to the spacing of the component groups after all the components of the previous group are removed from

INCONV. It is assumed that the first component group is properly located under CAMERA and does not require any belt translation. Software modules and functions have been developed for the various tasks and inherent processes associated with WIDGET assembly. The integration of these modules into a job plan is the responsibility of ODCAP and the workstation operator and is addressed in the following section.

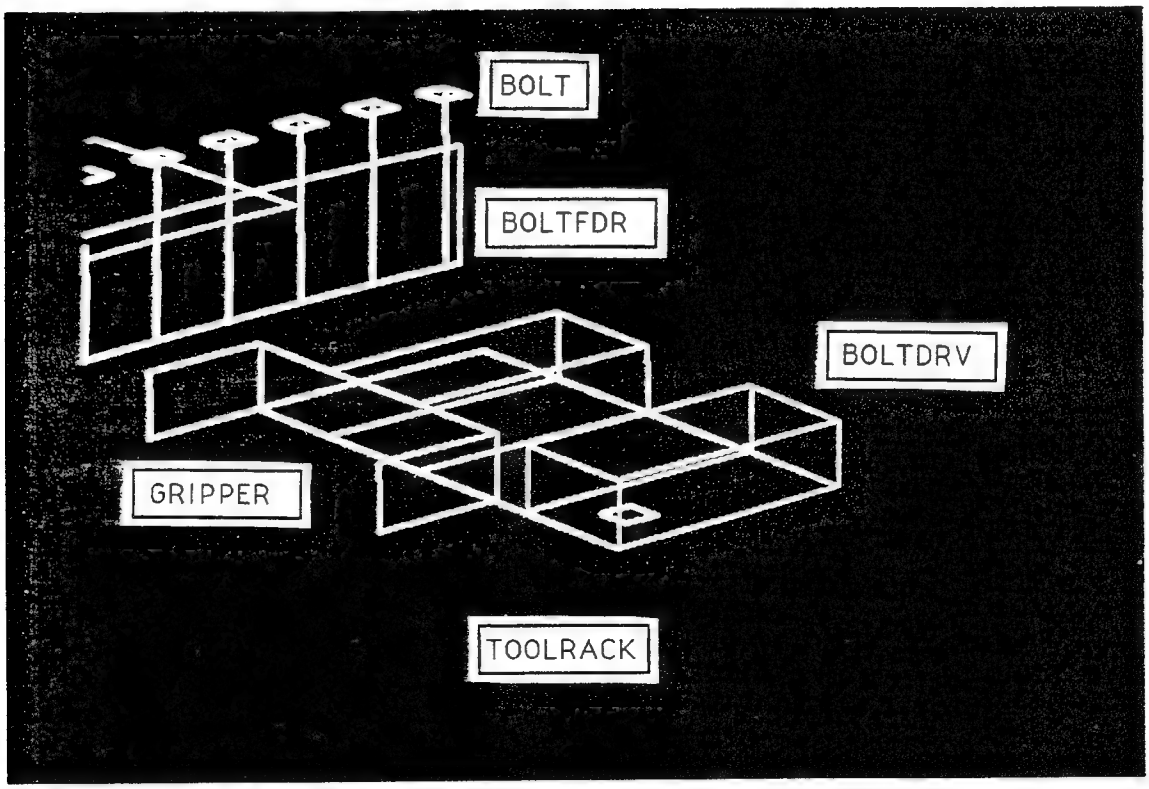
One additional device (affixed to BOLTDRV) labeled BDRVSEN acts as a binary sensor, detecting whether (or not) a BOLT fastener is attached to BOLTDRV. This device is not included in any task in the original job plan but plays a prominent role in the "replanning" example. Figure 5.1 includes two photographs of the simulated assembly workstation as it initially appears on the screen of the GS when ODCAP is activated. An overview of all workstation devices and objects is exhibited in Figure 5.1(a) while Figure 5.1(b) depicts (using a more narrow perspective than the first view) the tools GRIPPER and BOLTDRV situated on TOOLRACK as well as the BOLT fasteners held by BOLTFDR. Every device and object included in the WIDGET assembly example (except BDRVSEN) is shown and labeled in Figure 5.1.

ODCAP Job Planning

Initialization of the ODCAP software for the job assembly example, previously described, requires the insertion of initial global parameter data to the files INDEV.LST and



(a) Overall View



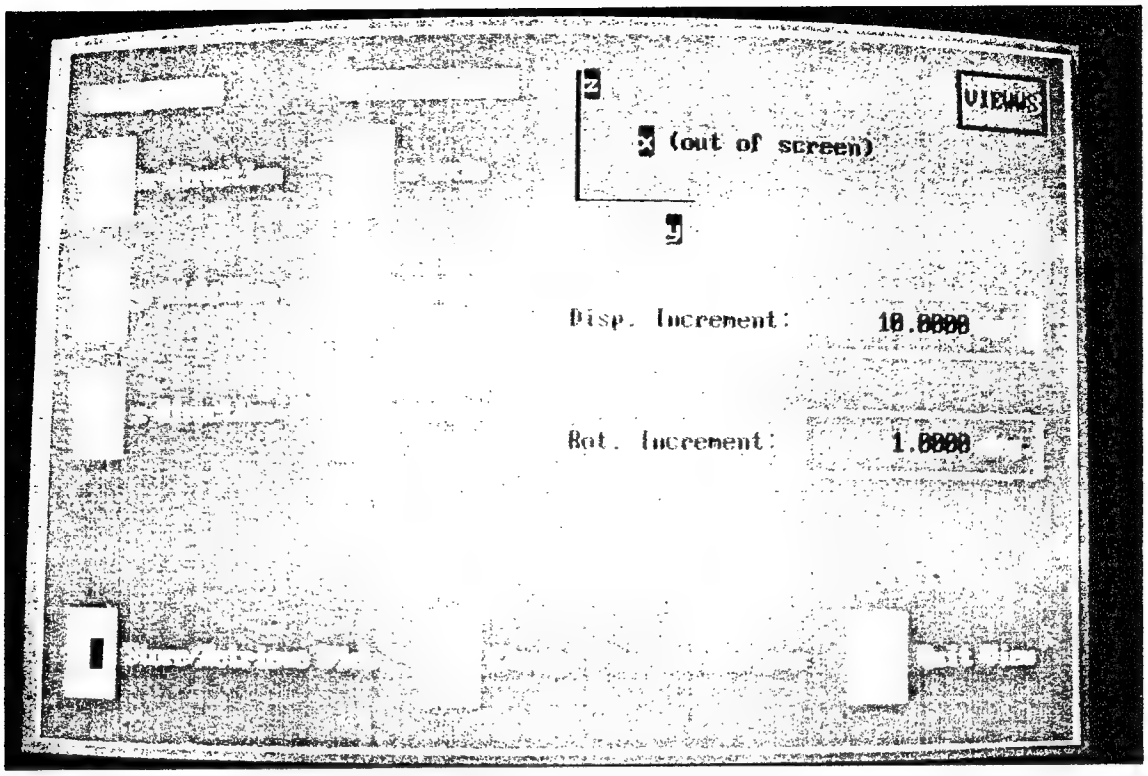
(b) TOOLRACK View

Figure 5.1. Assembly Workstation Simulation

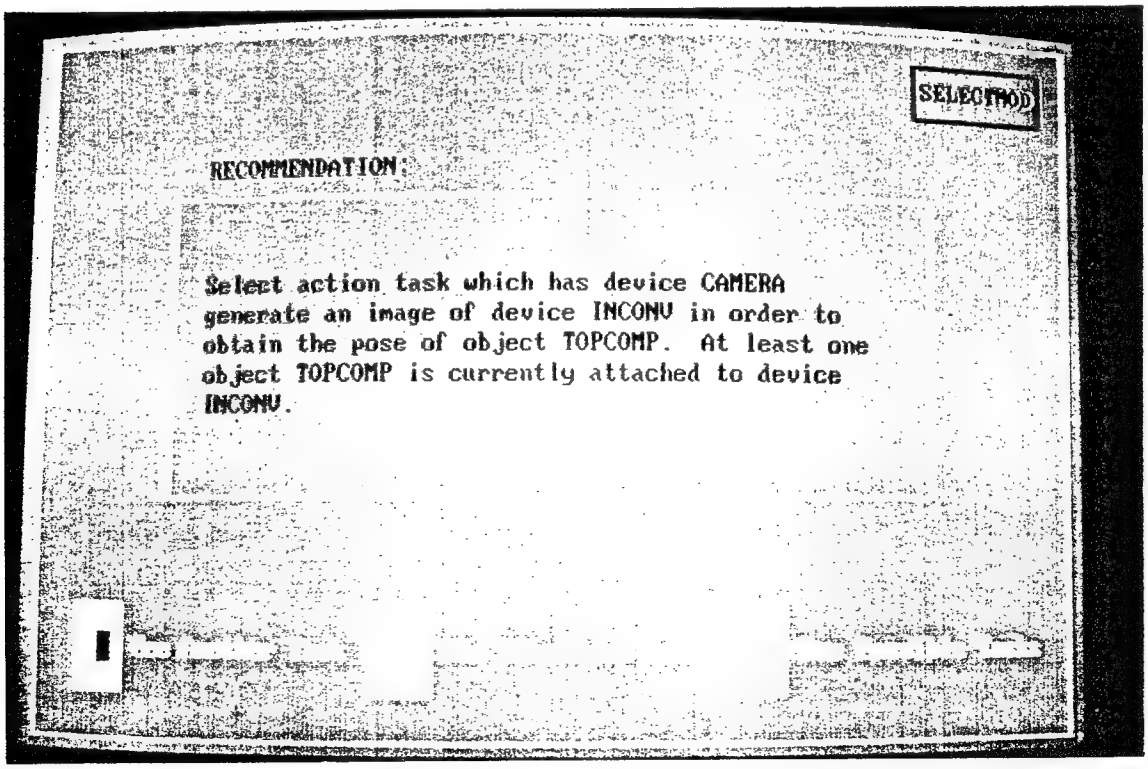
INOBS.LST (which reside in the \SIM subdirectory of the UIP's hard disk). Appendix G explains the procedures the workstation operator should perform to achieve the initial data specification. This duty is an important one since different initial device and object parameter settings may result in different job plans. ODCAP initialization also includes the execution of the program SETUP.EXE (in the UIP) which prompts the operator for the names of all the devices and objects present in the workstation. For the assembly example, the operator should enter the labels for the twelve devices and three objects discussed in the previous section. ODCAP automatically determines the contents of the three rule-bases based on the workstation configuration.

ODCAP's job planning algorithm is activated by the operator in the manner described in Appendix G. ODCAP queries the operator for the plan identification number; a value different from that of any existent job plan should be entered so that the algorithm can assume a regular planning mode. The screen display of the GS should resemble one of the workstation views shown in Figure 5.1, dependent on the viewing perspective. Figure 5.2(a) reproduces one of several menus that the UIP displays during ODCAP's interaction with the operator during his/her specification of a viewing perspective.

The algorithm continues with the operator's selection of the first task necessary to the achievement of the assembly job by the workstation. ODCAP displays recommended



(a) Viewing Perspective



(b) Explicit Module Recommendation

Figure 5.2. ODCAP UIP Menu Displays

task information on the UIP's screen as shown in Figure 5.2(b). The operator can either accept the recommendation or review additional recommendations; the example depicted in Figure 5.2(b) involves an action task in which CAMERA is used to generate an image of INCONV to determine position information on TOPCOMP objects. For the first task of the WIDGET assembly problem, the operator decides to accept ODCAP's recommendation that the GRIPPER tool should be retrieved from TOOLRACK and attached to the gripping device TOOLGRP. The module which corresponds with this task, labeled **get_tool()()**, contains eight different variable-structures in its list of input data. ODCAP activates the IRB in an attempt to ascertain implicit module(s) whose output(s) satisfy the input requirements of **get_tool()()**. If ODCAP fails in this endeavor, the operator will be prompted for a different task selection; otherwise, the algorithm continues with task simulation.

Task simulation for newly applied modules such as **get_tool()()** may require input from the operator relating to task execution. This information, once acquired by ODCAP, is written to the introductory data section and is referenced by function number and computer name. Three data items are needed for the application of **get_tool()()** to the plan, namely the approach and depart distances for TOOLGRP as it couples with GRIPPER and the speed of ROBOT during task performance. ODCAP automatically accesses whatever additional information it requires from the device- and

object-specific data files ($.DEV and %.OBS). This semi-autonomous procedure means that complex manufacturing information such as that related to the kinematics of a device acquiring an object can be linked to workstation resources instead of activities. The arduous chore of integrating the data associated with workstation activities to the hardware which implements those activities is shifted from the operator to the planning scheme. Operator input is only required for simple data items such as those mentioned in connection with **get_tool()()**.

ODCAP grants the operator one final chance to veto the insertion of the selected module prior to adding it to the plan. Figure 5.3 provides the Master Plan listing following the addition of the **get_tool()()** module. Four implicit modules are placed before **get_tool()()** to satisfy its input data requirements; none of these modules have input data requirements of their own. Every module is identified by a four character reference (functions utilize a similar five character reference) in which the first character denotes whether the module is explicit ("E") or implicit ("I") and the other characters represent the module's identification number. Modules are not numbered sequentially but by the order in which they are added to the plan. For the example plan of Figure 5.3, **intro_dev_swt_info()()** (I003) is included in the plan before the **intro_dev_var_info()()** module (I004). The planning algorithm continues with the graphics simulation of the designated tasks, which for **get_tool()()**

```
I004  intro_dev_var_info
      (ROBOT)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS)
I003  intro_dev_swt_info
      (TOOLGRP)
      (ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS)
I002  intro_dev_atc_info
      (TOOLGRP)
      (ndevatc:TOOLGRP:SS, devatci:TOOLGRP:SS)
I001  intro_dev_atc_info
      (TOOLRACK)
      (ndevatc:TOOLRACK:SS, devatci:TOOLRACK:SS)
E005  get_tool
      (GRIPPER, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
--->
```

Figure 5.3. Master Plan Listing after Step 001

will result in the attachment of GRIPPER to TOOLGRP and the subsequent removal of GRIPPER from TOOLRACK. The conclusion of the simulation marks the end of one planning cycle; each cycle is enumerated by a value known as the step number (Step 001 for **get_tool()()** in this example).

The introductory data section and the three subplan listings which correspond with the partial Master Plan of Figure 5.3 are shown in Figure 5.4 and Figure 5.5 respectively. The data files containing these plan items are created at the end of job planning by ODCAP; the contents of the files are quite large so only a portion of the total file is included in each of the figures. Process-level data structures in the introductory data section are referenced by function number, computer label, and structure name. The numerical values defined for the structures are listed beneath these references. The variable-structure "speed" listed under "0041 SS" in Figure 5.4, for example, corresponds with the output data item "speed" of the RC function intro_sginte()() identified at E0041 in Figure 5.5. The data structure "speed" contains a single integer value which for this particular function is specified with the value "12" (as indicated in Figure 5.4). The numerical values contained in the introductory data section are integrated with their data structures when the subplans are transformed into language code by the program generator.

An arrow, like the one shown at the bottom of the module listing in Figure 5.3, is utilized to track the

```
 .                                .
 .                                .
 .                                .
0026  SS
apptranstool
      0.0000       1.0000       0.0000       0.0000
      0.0000       0.0000       1.0000       0.0000
      1.0000       0.0000       0.0000     150.0000
apptranstlhr
      1.0000       0.0000       0.0000       0.0000
      0.0000       0.0000       1.0000     112.5000
      0.0000      -1.0000       0.0000       0.0000
0027  SS
devtrans
      1.0000       0.0000       0.0000    -100.0000
      0.0000       1.0000       0.0000     600.0000
      0.0000       0.0000       1.0000     330.0000
0029  SS
tlwrtrans
      1.0000       0.0000       0.0000       0.0000
      0.0000       1.0000       0.0000       0.0000
      0.0000       0.0000       1.0000       0.0000
0041  RC
speed
    12
0044  RC
speed
    12
0053  RC
speed
    12
0063  RC
countval
     2
0077  SS
numdevvar
     1
devvarinfo
ib                0.0000
0084  VC
cnvfxdtrans
      1.0000       0.0000       0.0000    -300.0000
      0.0000       0.0000       1.0000   -1800.0000
      0.0000      -1.0000       0.0000     300.0000
0086  VC
camtrans
      1.0000       0.0000       0.0000       0.0000
      0.0000      -1.0000       0.0000   -1100.0000
      0.0000       0.0000      -1.0000    1100.0000
 .                                .
 .                                .
 .                                .
```

Figure 5.4. Introductory Data Section (partial)

FCT #	VC LISTING	SS LISTING	RC LISTING
.	.	.	.
.	.	.	.
.	.	.	.
E0035			dis_mssge_RESP0 (mssge) (resp001)
E0036		if001 (resp001) ()	if001 (resp001) ()
E0037		asm_mssge_ADOU3 (traj1, linetraj2, 4) (mssge)	
E0038		xmit_mssge_ss_rc (mssge) ()	recv_mssge_ss_rc () (mssge)
E0039			dis_mssge_ADOU3 (mssge, 4) (traj, linetraj)
E0040			wait_robot_done () ()
E0041			intro_sginte () (speed)
E0042			move_robot_jtintrp (ROBOT, traj, speed) ()
E0043			wait_robot_done () ()
E0044			intro_sginte () (speed)
E0045			move_robot_strline (ROBOT, linetraj, speed) ()
E0046			wait_robot_done () ()
E0047		break () ()	break () ()
E0048		activate_switch (TOOLGRP, TGRPSTAT, ON) ()	
.	.	.	.
.	.	.	.
.	.	.	.

Figure 5.5. Subplan Listings (partial)

execution of plan sequences and is henceforth known as the pointer. The Master Plan after Step 001 has the pointer situated after **get_tool()()** indicating that the next module added to the plan by ODCAP and by the operator will be placed here. The operator selects a conditional task for Step 002 which implements three repetitions of WIDGET assembly. This task corresponds to the conditional module set **repeat###()():until_counter_done###()()** as depicted in the Master Plan listing after Step 002 in Figure 5.6. The operator is asked by ODCAP for the number of repetitions (three in this case) which the conditional module set is to perform. The pointer is situated between the modules **repeat001()()** and **until_counter_done001()()** since the first of three conditional repetitions is currently in effect. Appendix A provides complete information on the operation and usage of conditional tasks and modules.

The fact that an object's pose is known or unknown by the workstation is information which is held in one of the facts connected with the Explicit Rule-Base system (see Appendix D for fact and rule examples). ODCAP uses this information when it assesses its recommendations for the next task required for WIDGET assembly. The objects TOPCOMP and BOTCOMP situated on INCONV, as illustrated in Figure 5.1(a), have unknown positions and orientations. ODCAP recommends that a sensory action task be added to the task sequence to obtain object pose data (the UIP screen display for this recommendation is featured in Figure 5.2(b)). The

```
I004   intro_dev_var_info
       (ROBOT)
       (ndevvar:ROBOT:SS, devvari:ROBOT:SS)
I003   intro_dev_swt_info
       (TOOLGRP)
       (ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS)
I002   intro_dev_atc_info
       (TOOLGRP)
       (ndevatc:TOOLGRP:SS, devatci:TOOLGRP:SS)
I001   intro_dev_atc_info
       (TOOLRACK)
       (ndevatc:TOOLRACK:SS, devatci:TOOLRACK:SS)
E005   get_tool
       (GRIPPER, ROBOT, TOOLGRP, TOOLRACK,
        ndevvar:ROBOT:SS, devvari:ROBOT:SS,
        ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
        devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
        devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
       (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
        ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
        devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
        devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
E006   repeat001
       ()
       (i001:SS, count001:SS)
--->
E007   until_counter_done001
       (i001:SS, count001:SS)
       ()
```

Figure 5.6. Master Plan Listing after Step 002

simulation of this task will not affect the visual display of the workstation, but it will alter the global parameter values associated with the pose data for objects on INCONV. Figure 5.7 shows the module listing after Step 003. The explicit module **gen_conv_image_for_obs_info()()** performs the sensory task; the module requires input data (from the implicit module with identification number I008) related to the position of INCONV's belt on which the sensed objects are presently located.

Step 004 involves operator selection of a conditional task which alters the task execution sequence based on the condition that one TOPCOMP and one BOTCOMP constitute (or do not constitute) the component group currently located on INCONV. The module set which corresponds to this task is **if_widget_obs_on_dev###()():else###()():endif###()()** and is shown near the bottom of the Master Plan listing in Figure 5.8. The output data structures nobsatc:INCONV:VC and obsatci:INCONV:VC produced by the sensing action module applied in Step 003 (E009) contain pose data for all the components on INCONV which are positioned within CAMERA's field of view. The module set **if_widget_obs_on_dev###()(): else###()():endif###()()** uses this data to determine if the plan pointer should be placed before or after the module **else002()()** (E012). The first group of INCONV components for this planning example consists of one TOPCOMP and one BOTCOMP so the pointer (as indicated in Figure 5.8) would be inserted before the **else002()()** module. The implicit module

```
I008  intro_dev_var_info
      (INCONV)
      (ndevvar:INCONV:SS, devvari:INCONV:SS)
I004  intro_dev_var_info
      (ROBOT)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS)
I003  intro_dev_swt_info
      (TOOLGRP)
      (ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS)
I002  intro_dev_atc_info
      (TOOLGRP)
      (ndevatc:TOOLGRP:SS, devatci:TOOLGRP:SS)
I001  intro_dev_atc_info
      (TOOLRACK)
      (ndevatc:TOOLRACK:SS, devatci:TOOLRACK:SS)
E005  get_tool
      (GRIPPER, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
       (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
E006  repeat001
      ()
      (i001:SS, count001:SS)
E009  gen_conv_image_for_obs_info
      (CAMERA, INCONV, ndevvar:INCONV:SS,
       devvari:INCONV:SS)
      (ndevvar:INCONV:SS, devvari:INCONV:SS,
       nobsatc:INCONV:VC, obsatci:INCONV:VC)
--->
E007  until_counter_done001
      (i001:SS, count001:SS)
      ()
```

Figure 5.7. Master Plan Listing after Step 003

```
I008  intro_dev_var_info
      (INCONV)
      (ndevvar:INCONV:SS, devvari:INCONV:SS)
I004  intro_dev_var_info
      (ROBOT)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS)
I003  intro_dev_swt_info
      (TOOLGRP)
      (ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS)
I002  intro_dev_atc_info
      (TOOLGRP)
      (ndevatc:TOOLGRP:SS, devatci:TOOLGRP:SS)
I001  intro_dev_atc_info
      (TOOLRACK)
      (ndevatc:TOOLRACK:SS, devatci:TOOLRACK:SS)
E005  get_tool
      (GRIPPER, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
E006  repeat001
      ()
      (i001:SS, count001:SS)
E009  gen_conv_image_for_obs_info
      (CAMERA, INCONV, ndevvar:INCONV:SS,
       devvari:INCONV:SS)
      (ndevvar:INCONV:SS, devvari:INCONV:SS,
       nobsatc:INCONV:VC, obsatci:INCONV:VC)
I010  pass_conv_image_info
      (INCONV, nobsatc:INCONV:VC,
       obsatci:INCONV:VC)
      (nobsatc:INCONV:SS, obsatci:INCONV:SS)
E011  if_widget_obs_on_dev002
      (INCONV, nobsatc:INCONV:SS,
       obsatci:INCONV:SS)
      (nobsatc:INCONV:SS, obsatci:INCONV:SS)
--->
E012  else002
      (INCONV)
      ()
E013  endif002
      (INCONV)
      ()
E007  until_counter_done001
      (i001:SS, count001:SS)
      ()
```

Figure 5.8. Master Plan Listing after Step 004

pass_conv_image_info()() (I010) is introduced by ODCAP's Implicit Rule-Base system to transfer the data structures nobsatc:INCONV:VC and obsatci:INCONV:VC to the SS where they are renamed nobsatc:INCONV:SS and obsatci:INCONV:SS. The actions incorporated in this implicit task would only be performed when the plan is executed on-line.

The planning of the task sequence for the remainder of the assembly job proceeds in a similar fashion to that of the first four steps. The Master Plan for the entire WIDGET assembly job is listed in four parts in Figures 5.9 through 5.12. The module **pick_place_obs_to_dev()()** is utilized in two places (E015 and E016) to perform the robotic transfer of the two WIDGET components from INCONV to TABLE. The module **pick_insert_obs_to_asm()()** (E022 and E024) and the module **attach_fastener()()** (E023 and E025) perform the attachment of the BOLT fasteners to WIDGET. The final transfer of the finished WIDGET to the device OUTCONV is achieved by the **pick_place_asm_to_dev()()** (E029) module.

Following the module **else002()()** (E012) are modules which handle the situation when a component group on INCONV does not consist of one TOPCOMP and one BOTCOMP object. The module **pick_place_obs_to_dev()()** is used twice (E033 and E036), once to have ROBOT transport a BOTCOMP component from INCONV to the sink device REJBIN and the second time to move TOPCOMP to REJBIN. Both of these modules are supplied with the position and orientation information of all the components on INCONV sensed by CAMERA. Action modules such as

```
I032  intro_obs_atc_info
      (REJBIN)
      (nobsatc:REJBIN:SS, obsatci:REJBIN:SS)
I028  intro_asm_atc_info
      (OUTCONV)
      (nasmatc:OUTCONV:SS, asmatci:OUTCONV:SS)
I021  intro_obs_atc_info
      (BOLTDRV)
      (nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS)
I017  intro_asm_atc_info
      (TABLE)
      (nasmatc:TABLE:SS, asmatci:TABLE:SS)
I014  intro_obs_atc_info
      (TABLE)
      (nobsatc:TABLE:SS, obsatci:TABLE:SS)
I008  intro_dev_var_info
      (INCONV)
      (ndevvar:INCONV:SS, devvari:INCONV:SS)
I004  intro_dev_var_info
      (ROBOT)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS)
I003  intro_dev_swt_info
      (TOOLGRP)
      (ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS)
I002  intro_dev_atc_info
      (TOOLGRP)
      (ndevatc:TOOLGRP:SS, devatci:TOOLGRP:SS)
I001  intro_dev_atc_info
      (TOOLRACK)
      (ndevatc:TOOLRACK:SS, devatci:TOOLRACK:SS)
E005  get_tool
      (GRIPPER, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
E006  repeat001
      ()
      (i001:SS, count001:SS)
E009  gen_conv_image_for_obs_info
      (CAMERA, INCONV, ndevvar:INCONV:SS,
       devvari:INCONV:SS)
      (ndevvar:INCONV:SS, devvari:INCONV:SS,
       nobsatc:INCONV:VC, obsatci:INCONV:VC)
I010  pass_conv_image_info
      (INCONV, nobsatc:INCONV:VC,
       obsatci:INCONV:VC)
      (nobsatc:INCONV:SS, obsatci:INCONV:SS)
```

Figure 5.9. Final Master Plan Listing (Part I)

```
E011  if_widget_obs_on_dev002
      (INCONV, nobsatc:INCONV:SS,
       obsatci:INCONV:SS)
      (nobsatc:INCONV:SS, obsatci:INCONV:SS)
E015  pick_place_obs_to_dev
      (BOTCOMP, ROBOT, TOOLGRP, GRIPPER,
       INCONV, TABLE, ndevvar:ROBOT:SS,
       devvari:ROBOT:SS, nobsatc:INCONV:SS,
       obsatci:INCONV:SS, nobsatc:TABLE:SS,
       obsatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:INCONV:SS, obsatci:INCONV:SS,
       nobsatc:TABLE:SS, obsatci:TABLE:SS)
E016  pick_place_obs_to_dev
      (TOPCOMP, ROBOT, TOOLGRP, GRIPPER,
       INCONV, TABLE, ndevvar:ROBOT:SS,
       devvari:ROBOT:SS, nobsatc:INCONV:SS,
       obsatci:INCONV:SS, nobsatc:TABLE:SS,
       obsatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:INCONV:SS, obsatci:INCONV:SS,
       nobsatc:TABLE:SS, obsatci:TABLE:SS)
E018  mate_obs_to_obs
      (TOPCOMP, BOTCOMP, ROBOT, TOOLGRP,
       GRIPPER, TABLE, WIDGET,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:TABLE:SS, obsatci:TABLE:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:TABLE:SS, obsatci:TABLE:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
E019  put_back_tool
      (GRIPPER, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
E020  get_tool
      (BOLTDRV, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
```

Figure 5.10. Final Master Plan Listing (Part II)

```
E022  pick_insert_obs_to_asm
      (BOLT, WIDGET, ROBOT, TOOLGRP,
       BOLTDRV, BOLTFDR, TABLE, BOTCOMP,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
E023  attach_fastener
      (BOLT, WIDGET, ROBOT, TOOLGRP,
       BOLTDRV, TABLE, ndevvar:ROBOT:SS,
       devvari:ROBOT:SS, nobsatc:BOLTDRV:SS,
       obsatci:BOLTDRV:SS, nasmatc:TABLE:SS,
       asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
E024  pick_insert_obs_to_asm
      (BOLT, WIDGET, ROBOT, TOOLGRP,
       BOLTDRV, BOLTFDR, TABLE, BOTCOMP,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
E025  attach_fastener
      (BOLT, WIDGET, ROBOT, TOOLGRP,
       BOLTDRV, TABLE, ndevvar:ROBOT:SS,
       devvari:ROBOT:SS, nobsatc:BOLTDRV:SS,
       obsatci:BOLTDRV:SS, nasmatc:TABLE:SS,
       asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
E026  put_back_tool
      (BOLTDRV, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
E027  get_tool
      (GRIPPER, ROBOT, TOOLGRP, TOOLRACK,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevswt:TOOLGRP:SS, devswti:TOOLGRP:SS,
       devatci:TOOLGRP:SS, ndevatc:TOOLGRP:SS,
       devatci:TOOLRACK:SS, ndevatc:TOOLRACK:SS)
```

Figure 5.11. Final Master Plan Listing (Part III)

```
E029  pick_place_asm_to_dev
      (WIDGET, BOTCOMP, ROBOT, TOOLGRP,
       GRIPPER, TABLE, OUTCONV,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS,
       nasmatc:OUTCONV:SS, asmatci:OUTCONV:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS,
       nasmatc:OUTCONV:SS, asmatci:OUTCONV:SS)
E012  else002
      (INCONV)
      ()
E030  while_object_on_device003
      (BOTCOMP, INCONV, nobsatc:INCONV:SS,
       obsatci:INCONV:SS)
      (nobsatc:INCONV:SS, obsatci:INCONV:SS)
E033  pick_place_obs_to_dev
      (BOTCOMP, ROBOT, TOOLGRP, GRIPPER,
       INCONV, REJBIN, ndevvar:ROBOT:SS,
       devvari:ROBOT:SS, nobsatc:INCONV:SS,
       obsatci:INCONV:SS, nobsatc:REJBIN:SS,
       obsatci:REJBIN:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:INCONV:SS, obsatci:INCONV:SS,
       nobsatc:REJBIN:SS, obsatci:REJBIN:SS)
E031  endwhile003
      (BOTCOMP, INCONV)
      ()
E034  while_object_on_device004
      (TOPCOMP, INCONV, nobsatc:INCONV:SS,
       obsatci:INCONV:SS)
      (nobsatc:INCONV:SS, obsatci:INCONV:SS)
E036  pick_place_obs_to_dev
      (TOPCOMP, ROBOT, TOOLGRP, GRIPPER,
       INCONV, REJBIN, ndevvar:ROBOT:SS,
       devvari:ROBOT:SS, nobsatc:INCONV:SS,
       obsatci:INCONV:SS, nobsatc:REJBIN:SS,
       obsatci:REJBIN:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:INCONV:SS, obsatci:INCONV:SS,
       nobsatc:REJBIN:SS, obsatci:REJBIN:SS)
E035  endwhile004
      (TOPCOMP, INCONV)
      ()
E013  endif002
      (INCONV)
      ()
E037  move_belt_user
      (INCONV, ndevvar:INCONV:SS,
       devvari:INCONV:SS)
      (ndevvar:INCONV:SS, devvari:INCONV:SS)
E007  until_counter_done001
      (i001:SS, count001:SS)
      ()
```

Figure 5.12. Final Master Plan Listing (Part IV)

pick_place_obs_to_dev()() contain conditional functions which prevent the module's task from being carried out if workstation conditions are unfavorable. For example, if an improper component group consisted of a single BOTCOMP object whose location was not within ROBOT's workspace, the **pick_place_obs_to_dev()()** module at E033 would yield no workstation activity. The module would terminate itself automatically once it determined from the input data that the desired robotic pick-and-place task was unachievable. The modules between **else002()()** (E012) and **endif002()()** (E013) can only be planned by ODCAP and by the operator if the list of simulated sourced objects includes component groups which do not consist of one TOPCOMP and one BOTCOMP. Planning always follows the course of simulated workstation activity, implying that any workstation conditions which can be simulated can be dealt with during planning.

The module **move_belt_user()()** (E037) commands the belt on the device INCONV to advance the next component group to a location under CAMERA. The workstation operator knows the approximate distance between incoming groups so he/she can enter this data when queried by ODCAP. When the last module in the Master Plan (**until_counter_done001()()** at E007) is reached during planning, the pointer which tracks the task sequence will be positioned (by ODCAP) after the module **until_counter_done001()()** only if all the modules between **repeat001()()** and **until_counter_done001()()** have been executed three times. If only one or two repetitions of these

modules have been carried out, the pointer is inserted into the plan immediately after the **repeat001()()** module. The execution of previously-planned modules which can result from conditional module sets (only) such as **repeat###()(): until_counter_done###()()** is conducted automatically by ODCAP using all the task specifications that the operator supplied when the module was initially installed.

The operator can conclude a planning session without concern for producing an unworkable plan since every module within a conditional module set is added to the plan whenever the first module in the set is selected for plan inclusion. ODCAP also prevents the improper nesting of multiple conditional sets from occurring by ensuring that new module sets are inserted entirely within existent conditional module sets. The pointer is removed from the Master Plan listing when the operator indicates that the plan is complete when he/she ends the planning session.

Replanning Procedures

The example job whose plan was developed in the previous section can be "replanned" using ODCAP to add new tasks in a nearly identical manner to that conducted during regular planning. The operator must specify the old plan identification number when queried by ODCAP after the software is activated. The planning software will automatically recognize that a plan exists for that number and will allow the operator to select the replanning mode. Existent Master

Plan modules, whose corresponding tasks are simulated during replanning, will automatically access the input data they require from the introductory data section. The source object information in INOBS.LST is needed for replanning and can be altered prior to replanning to determine how an existent plan handles different situations for objects entering the workstation.

ODCAP does not ask the operator for any information related to the execution of a previously-planned task. The plan modules are executed in the same order as listed in the Master Plan with conditional module sets affecting the task sequence just as they did during regular planning. Following an existent module's execution, ODCAP inquires from the operator about his/her intention to insert new explicit modules into the Master Plan. If the operator responds affirmatively, ODCAP will perform the same procedures as used during planning for selecting and applying new modules except that these new modules can be placed virtually anywhere in the old Master Plan. Replanning is demonstrated with the use of an error recovery situation affecting the WIDGET assembly job. The error stems from the condition that BOLTFDR may not be furnished with enough bolts (at least six are needed) for the assembly of three WIDGETs. The workstation should halt activity once all the BOLT fasteners supplied by BOLTFDR have been used in WIDGET manufacture, even if this termination results in an incomplete WIDGET assembly.

The sensor device BDRVSEN is utilized to detect when BOLTFDR runs out of BOLTs by determining if the tool BOLTDRV has correctly retrieved a BOLT fastener from BOLTFDR. The explicit module **sense_fastener_presence()()** performs this object detection task while the conditional module set **if_no_obs_on_dev###()():endif###()()** uses the information from the sensing task to alter the plan sequence so that error recovery tasks can be performed. The only error recovery module associated with this particular situation is **halt_activity()()** which shuts down the workstation when executed. The operator with ODCAP's assistance would insert these three modules after **pick_insert_obs_to_asm()()** (E022) in the manner shown in the module listing of Figure 5.13. The new modules are marked with module identification numbers greater in value than that of the last module (E037) in the existent Master Plan.

The added modules prevent a non-existent BOLT fastener from being driven into TOPCOMP and BOTCOMP by BOLTDRV during execution of the **attach_fastener()()** module (E023). The insertion of these modules requires the completion of three cycles of ODCAP's planning algorithm. The same three modules can be inserted into the Master Plan after the second occurrence of **pick_insert_obs_to_asm()()** (E024) to effect an identical error recovery solution for the second BOLT used in WIDGET assembly. Testing of these new modules requires that the operator alter the data in INOBS.LST so that

```
 .                    .
 .                    .
 .                    .
E022  pick_insert_obs_to_asm
      (BOLT, WIDGET, ROBOT, TOOLGRP,
       BOLTDRV, BOLTFDR, TABLE, BOTCOMP,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
E038  sense_fastener_presence
      (BOLTDRV, BDRVSEN)
      (nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS)
E039  if_no_obs_on_dev005
      (BOLTDRV, nobsatc:BOLTDRV:SS,
       obsatci:BOLTDRV:SS)
      (nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS)
E041  halt_activity
      ()
      ()
E040  endif005
      (BOLTDRV)
      ()
E023  attach_fastener
      (BOLT, WIDGET, ROBOT, TOOLGRP,
       BOLTDRV, TABLE, ndevvar:ROBOT:SS,
       devvari:ROBOT:SS, nobsatc:BOLTDRV:SS,
       obsatci:BOLTDRV:SS, nasmatc:TABLE:SS,
       asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
E024  pick_insert_obs_to_asm
      (BOLT, WIDGET, ROBOT, TOOLGRP,
       BOLTDRV, BOLTFDR, TABLE, BOTCOMP,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       nobsatc:BOLTDRV:SS, obsatci:BOLTDRV:SS,
       nasmatc:TABLE:SS, asmatci:TABLE:SS)
 .                    .
 .                    .
 .                    .
```

Figure 5.13. Module Listings for Replanning Example

BOLTFDR sources an insufficient number of BOLTs to the workstation. The planning and replanning demonstrations discussed in this chapter are available on videotape from the Mechanical Engineering Department, Clemson University.

CHAPTER VI

APPLICATION TO APPAREL MANUFACTURING

Introduction

The need to enhance the manufacturing of apparel shirt collars through the use of automation technology was expressed in Chapter I. A manufacturing workstation known as the Apparel Assembly Workstation (AAW) was introduced at that time which is capable of performing two critical shirt collar assembly operations, referred to as turning and pressing. The AAW and the assembly tasks associated with it provide a real-world manufacturing environment for the testing of the hierarchical control and planning scheme developed in this research. Modules and functions have been produced (similar to those software elements created for the WIDGET assembly problem) which direct the performance of AAW tasks and processes respectively. The planning of AAW activities is conducted by ODCAP and by an operator as demonstrated by the WIDGET assembly example of the previous chapter. AAW job planning, however, includes the additional step of converting the function sequences (subplans) into actual control programs which command physical workstation activity.

The device and computer resources constituting the AAW are described and are shown to be consistent with the generic workstation configuration developed in Chapter III.

The collar turning and pressing tasks as performed by the AAW are explained and related to their corresponding program modules. A complete Master Plan listing is provided for the collar turning and pressing job. The chapter concludes with discussions on the generation of control programs for the AAW computers and on the on-line operation of these programs.

Apparatus

The AAW is assembled around a single four-degree-of-freedom AdeptOne SCARA robot which performs the loading (and unloading) of single collars into specially designed turning and pressing machines. Two PC/ATs and one Adept robot controller comprise the AAW control computers. The labels of Robot Controller (RC), Vision Controller (VC), and System Supervisor (SS), used for the WIDGET assembly workstation computers, are adopted for the AAW computers. The RC directs robot motion while the VC takes in data from vision sensors (cameras) and analyses it. All AAW actuators (except the robot) as well as many sensors are controlled by the SS. The AAW's configuration of devices about these three computers is illustrated in Figure 6.1 and conforms to the general workstation configuration depicted in Figure 3.7 (see page 40) with n equal to two. Communication between the SS and its two subordinates, the RC and the VC, is

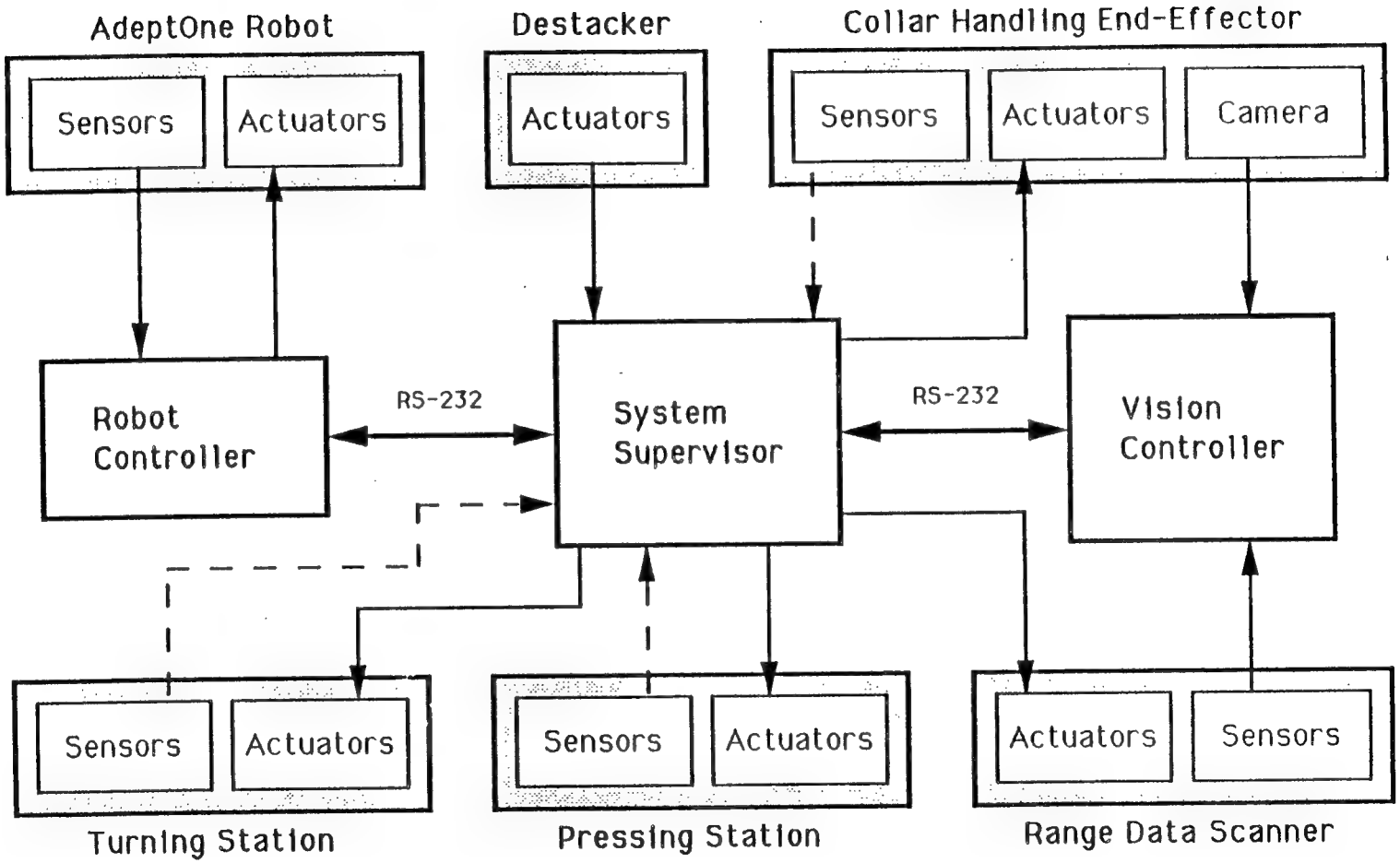


Figure 6.1. AAW Configuration

achieved with the use of RS-232 serial interfaces. Appendix H provides information regarding system communication and real-time device operation issues.

The proposed manner by which devices are classified (see the taxonomy chart in Figure 4.2 on page 78) does not permit a single device to be both an actuator and a sensor. Equipment such as the turning and pressing machines which incorporate both device sensors and actuators are referred to as "stations". The sensors for the turning and pressing stations as well as the collar handling end-effector are connected to the SS by dashed lines in Figure 6.1. These lines denote that the sensing devices play no part in the manufacturing operations conducted by the AAW. They are, however, utilized for the calibration of system actuators. Only the vision sensors linked to the VC (and the sensors in the robot used by the RC for determining arm position) have active roles in some AAW activities. The operation and composition of the AAW devices, which were designed exclusively for collar turning and pressing, are defined. The development of the AAW was carried out by a research team; those team members responsible for devising the AAW hardware are explicitly referenced in the device descriptions.

The "turning device" indicates a single device constituted from the mechanical actuators included in the turning station. An isometric depiction of the turning device is provided in Figure 6.2 [67]. The device inverts both collar points (see Figure 1.1 on page 5) with a single motion of

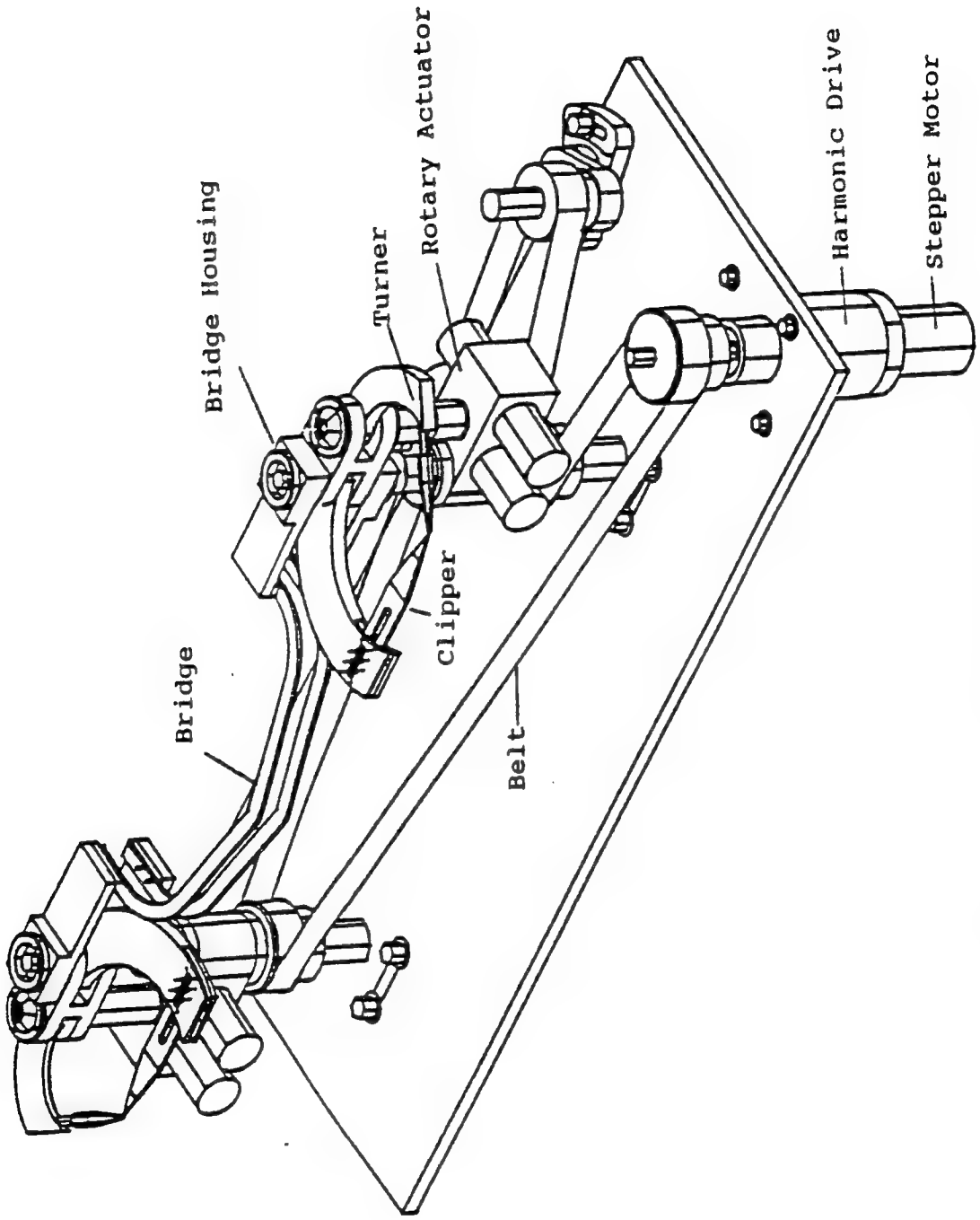


Figure 6.2. Turning Device (from [67])

its stepper motor actuator. Belt-driven components take hold of the collar by its points during turning. The unturned collar is forced through a stationary member known as the bridge which performs the reversal of the collar's upper and lower plies. The turning device can be loaded manually or robotically and does not require the reloading of a "half-turned" collar as is the case for most industrial collar turning machines which invert collars one point at a time.

The pressing device (a term incorporating all pressing station actuators) consists of mechanisms which insert thin blades into a turned collar to stretch it out prior to being pressed [68]. The blades are fabricated in the shape of a flattened collar. The mechanisms which insert these blades are designed such that the lower and upper plies are pulled taut through the application of opposing forces on the collar points. The pressing station uses a vacuum table to prevent the lower ply from shifting as the blades are inserted. The operation of the pressing device is analogous to the placement of a human hand inside a glove. The alignment of the stitch line (known as the seam) to the edges of the inserted blades is performed by linear actuators, which pull the collar's top ply in a direction perpendicular to the stitch line. The pressing device translates the stretched collar to a table where it is pressed (an operation performed manually with a hand iron) following seam

alignment and subsequently released from the inserted blades.

Unturned collars are sourced to the AAW by a device called the destacker. The collars are initially loaded (manually) into this device in stacked bundles. The destacker uses claw-like pickers to grasp and lift the topmost collar in the stack away from the other stacked collars. The weight of the lower ply causes the collar plies to separate when the upper ply is held by the destacker pickers.

A collar handling end-effector developed for the AAW can acquire collars from the destacker (and the turning device) and load them into the turning and pressing devices [69]. Figure 6.3 depicts the end-effector which utilizes two grippers to take hold of the collar's upper ply at two locations, in the same manner as a human operator would with two hands. The end-effector keeps the collar in a horizontal configuration during device loading and unloading and possesses a pitch actuator which can reorient the collar about a horizontal axis. Photographs of an unturned collar undergoing loading into the turning device and a turned collar being positioned for pressing blade insertion are included in Figures 6.4(a) and 6.4(b) respectively. A vision sensor (camera) is placed onboard the end-effector to assist with the pressing device's seam alignment operations.

The AAW includes a device group known as the Range Data Scanner (RDS) which employs laser-striping techniques to

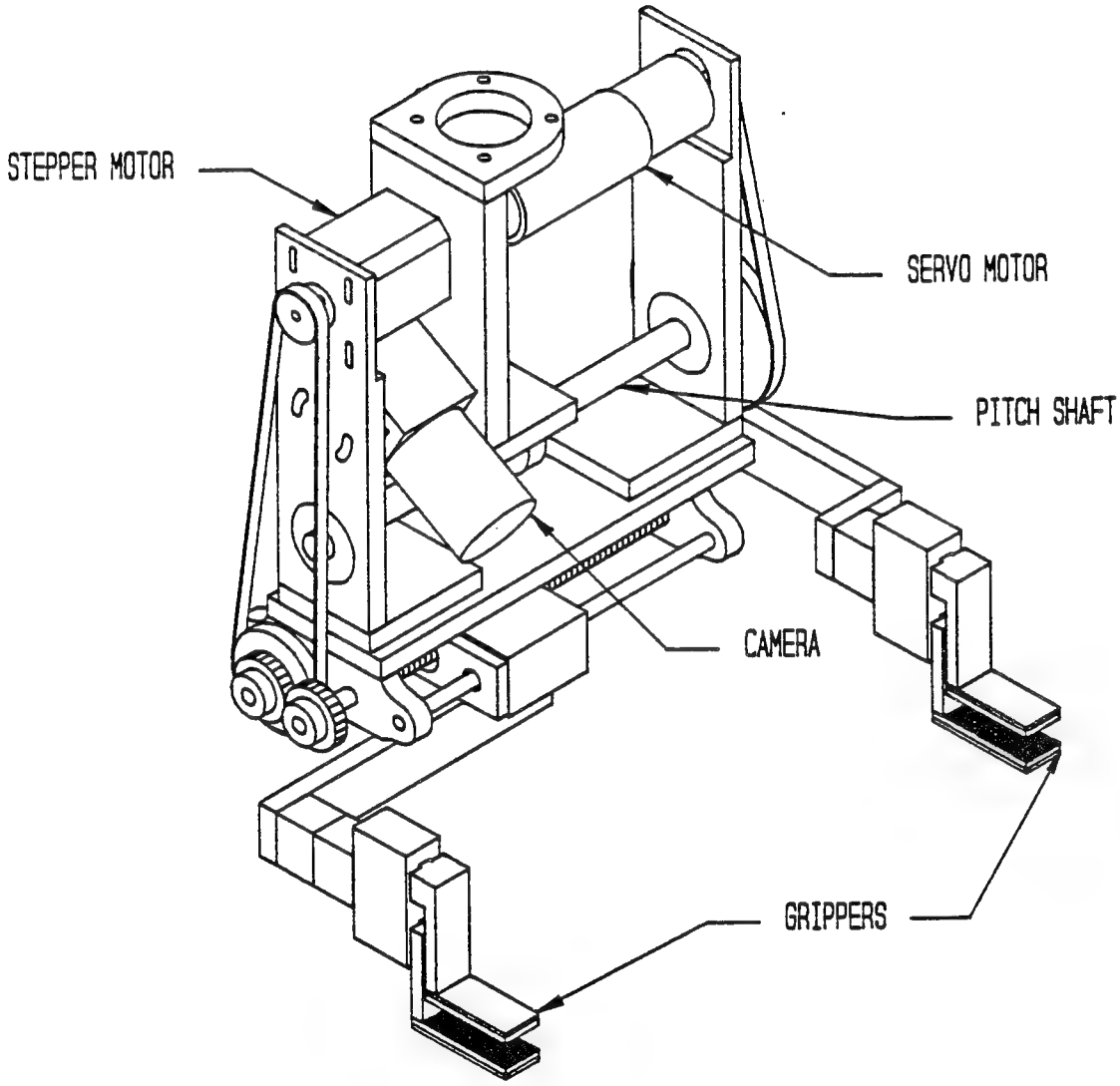
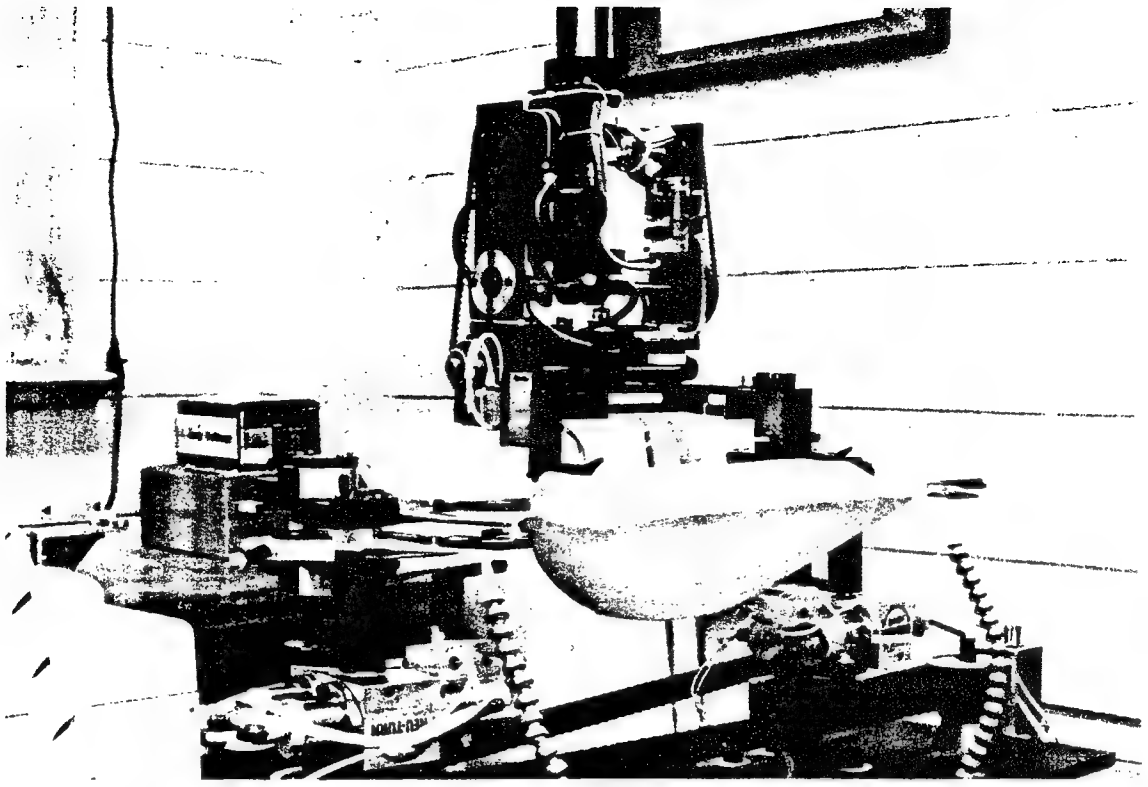
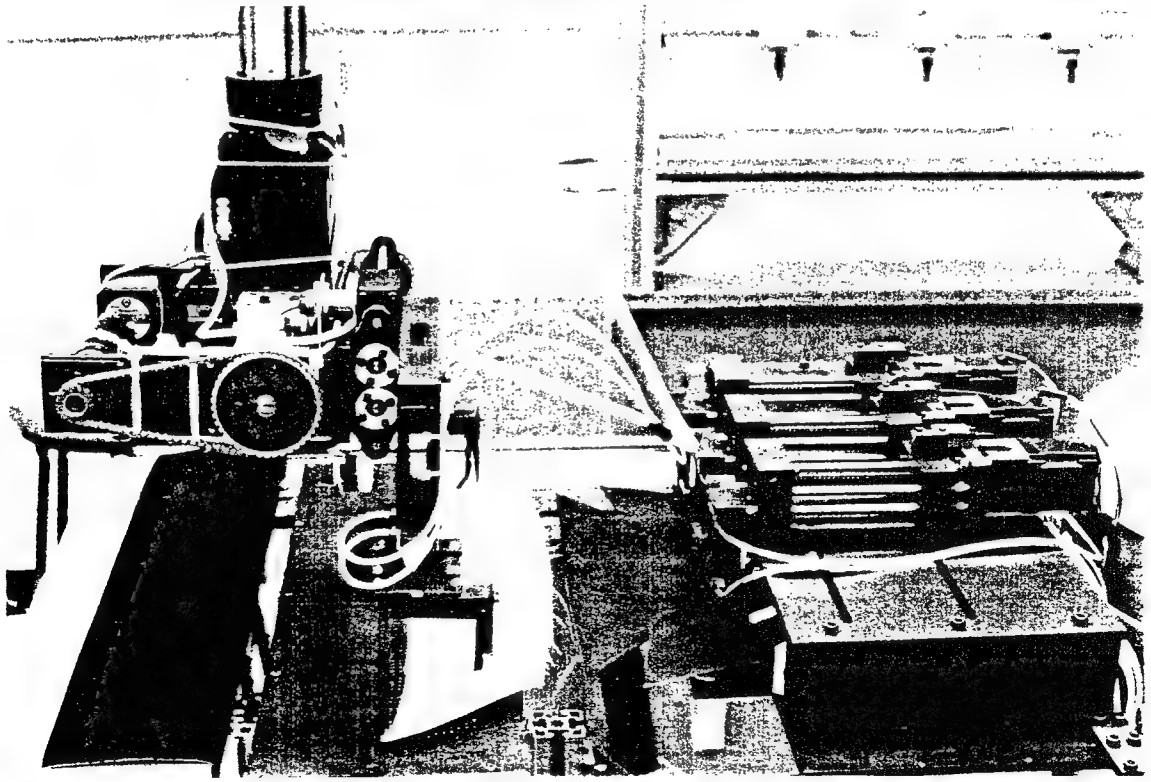


Figure 6.3. Collar Handling End-Effector (from [69])



(a) Turning Device loaded with Unturned Collar



(b) Pressing Device loaded with Turned Collar

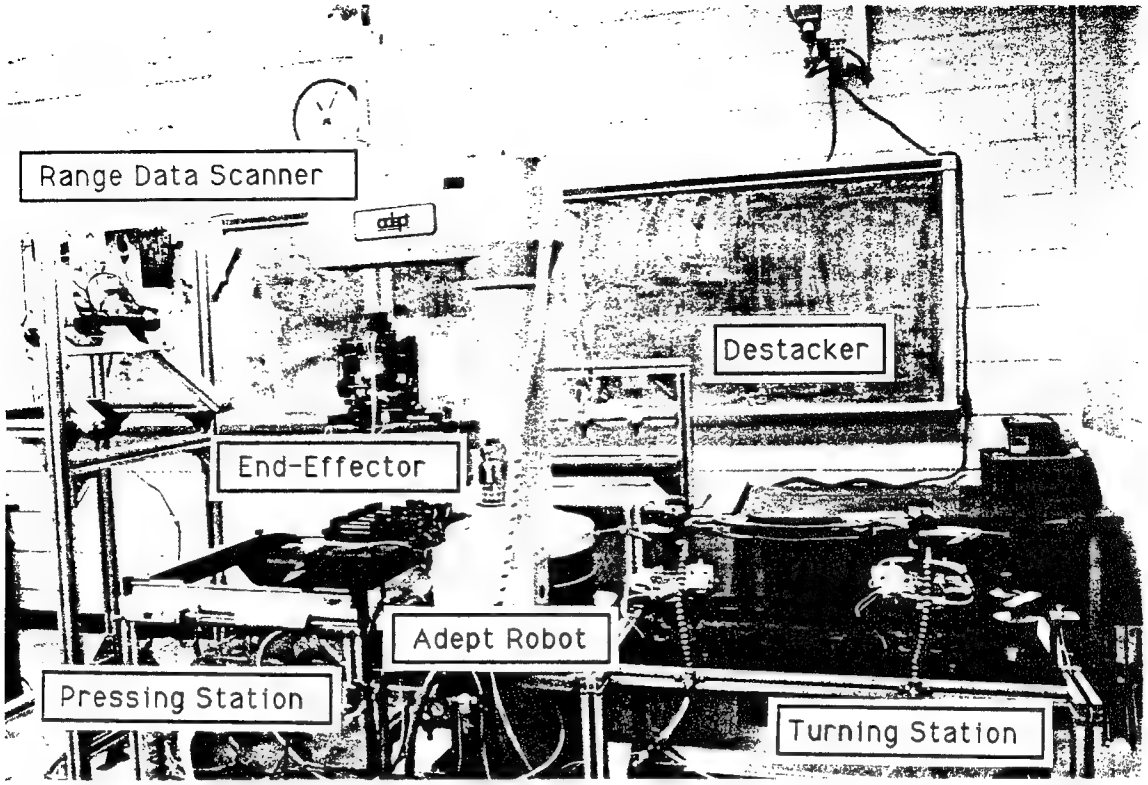
Figure 6.4. Device Loading

determine the three-dimensional coordinates of the two collar points before the turned collar is loaded by the robot with the end-effector into the pressing device [70]. The RDS projects a sheet of laser-generated light onto the collar by adjusting the light's angle of reflection off a revolving mirror driven by a rotary actuator. A single camera (situated in a location adjacent to the laser) generates an image of the light stripe produced when the laser sheet intersects the collar. Analyses involving triangulation are performed by VC software to calculate the collar point coordinates from the image data. These coordinate values affect the robot kinematics associated with the loading of the collar into the pressing device because they define the geometric relationship between the grasped collar and the end-effector. A photograph showing all AAW devices is provided in Figure 6.5(a) while Figure 6.5(b) exhibits the three AAW control computers.

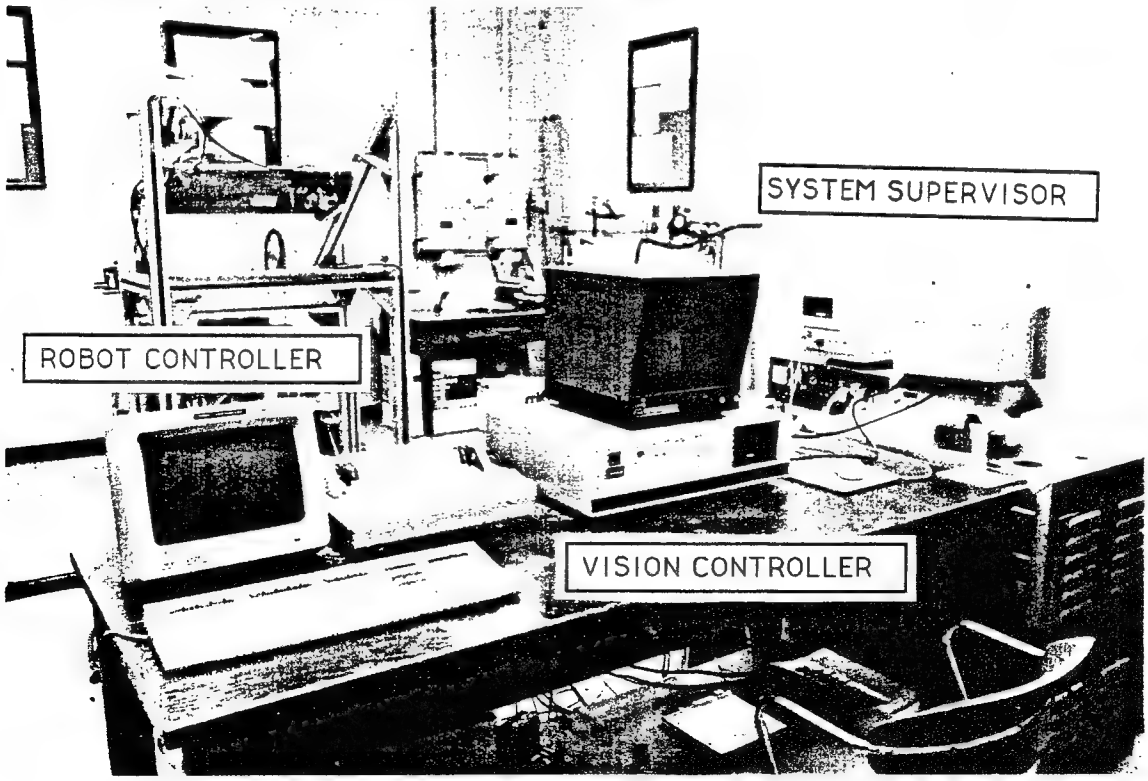
Experimental Results

The tasks which the AAW must perform to turn and press one apparel shirt collar are defined in the following sequential list.

1. Robot end-effector acquires an unturned collar from the destacker device.
2. Robot end-effector loads unturned collar into the turning device which inverts the collar points.
3. Robot end-effector reacquires turned collar from the turning device.



(a) Hardware Configuration



(b) Control Computers

Figure 6.5. Apparel Assembly Workstation

4. Robot end-effector transports collar to a location in front of the RDS so that laser-striping analyses can be performed and the coordinate values of the collar points can be determined.

5. Robot end-effector loads turned collar into the pressing device which inserts its pressing blades into the opened collar.

6. Pressing device actuators align the collar seam with the edges of the blades following the visual analysis of a collar image generated by the end-effector camera.

7. Pressing device actuators translate collar to the pressing table where it is pressed (by hand iron) and the blades are retracted from the collar.

The proceeding task sequence was implemented on the AAW using control programs developed without the assistance of the ODCAP software package. Two programs, one for the SS and another for the VC, were produced using Microsoft C while a third program was developed in "V+", the language adopted by Adept for controlling their robots. The creation of these programs required the utilization of numerous communication and synchronization routines, which transfer data between and coordinate the actions of the AAW computers respectively. These routines prevented the programmer from developing each program in isolation from the other two because communication and synchronization functions in one computer's software require counterparts in one (or two) of the other computer's software. The development of the AAW control programs which effect automated collar turning and pressing was a difficult undertaking requiring considerable time and tedious effort.

The ODCAP software package was applied to the AAW collar turning and pressing job to verify the proposed workstation control and planning scheme's usefulness to real-world manufacturing. Control programs similar to those developed without ODCAP were generated using the planning procedures detailed in Chapter IV. The seven step collar assembly task sequence is simplified through the removal of the two tasks (numbered 4 and 6) which require visual processing. This simplification enhances the implementation example by reducing the number of devices which need controlling and by focusing attention on the robotic loading/unloading and operation of the turning and pressing devices. Only two control programs, one for the SS and the other for the RC, must be generated to implement the five remaining tasks in the prescribed assembly sequence.

Three modules are created to direct all the activities associated with the five assembly tasks (numbered 1, 2, 3, 5, and 7). The module **unload_device()()** directs device unloading in tasks 1 and 3 with FINITE-STATEs used to specify the device (either the destacker or the turning device) which is to be unloaded. Tasks 2 and 5 are similarly performed by the explicit module **load_and_activate_device()()**. The module **press_obs()()** is developed to carry out the actions related to the final task in the sequence (task 7). The remaining sections of this chapter concern the arrangement of these modules into an AAW job plan by ODCAP and by the AAW operator. The execution of this plan on the AAW

hardware is discussed with information provided on ODCAP's usefulness in planning AAW activities.

AAW Job Planning

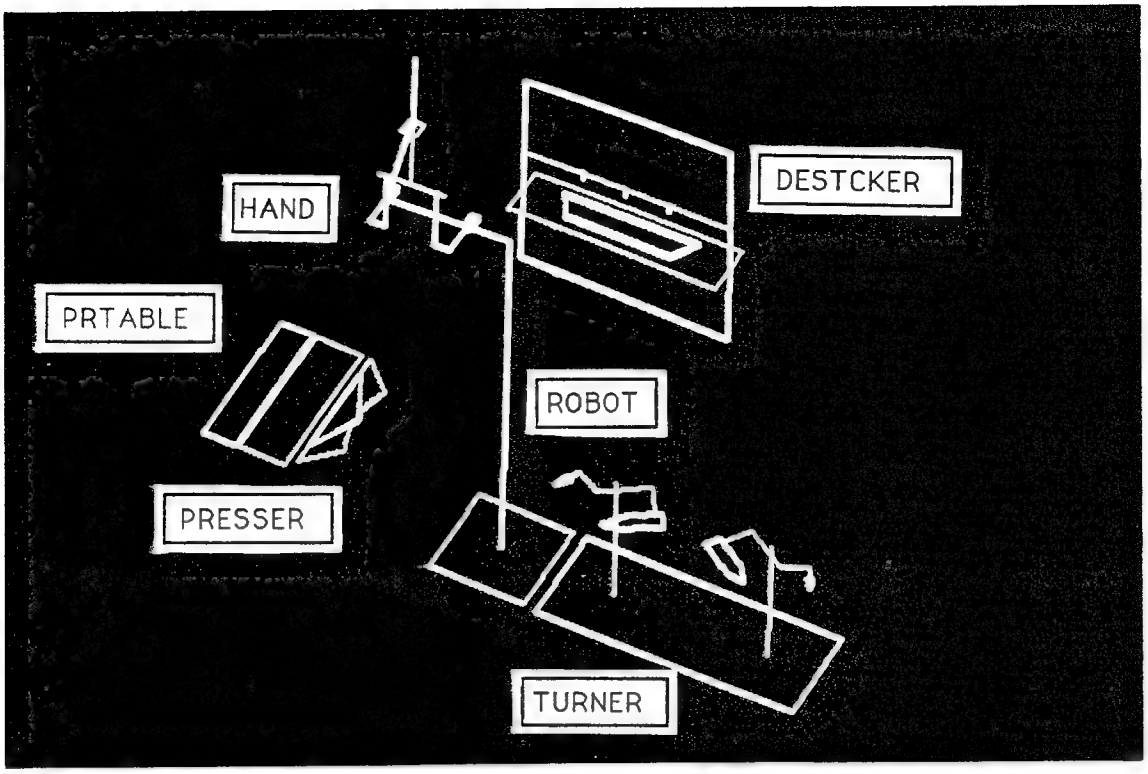
Job planning for the AAW proceeds in an identical manner to that demonstrated by the WIDGET assembly example, discussed in the previous chapter. Due to the simplicity of the proposed AAW collar turning and pressing job (consisting of only five action tasks), no additional rules are devised for either of ODCAP's two explicit rule-bases, the ENSFRB and the EMRRB. ODCAP provides task recommendations based on explicit rule-base analyses. If these rule-bases fail to provide task recommendations, the operator is still permitted to select new tasks corresponding to the modules which ODCAP determines are "possible". The list of possible modules, generated by ODCAP for every cycle of the planning algorithm, is never lengthy for the AAW example because the example only involves a few modules and a limited number of devices (six) and objects (one).

ODCAP initialization requires the operator to input the names of the six devices and one object which constitute the AAW resources. The following list provides the FINITE-STATE labels for all AAW device and object resources:

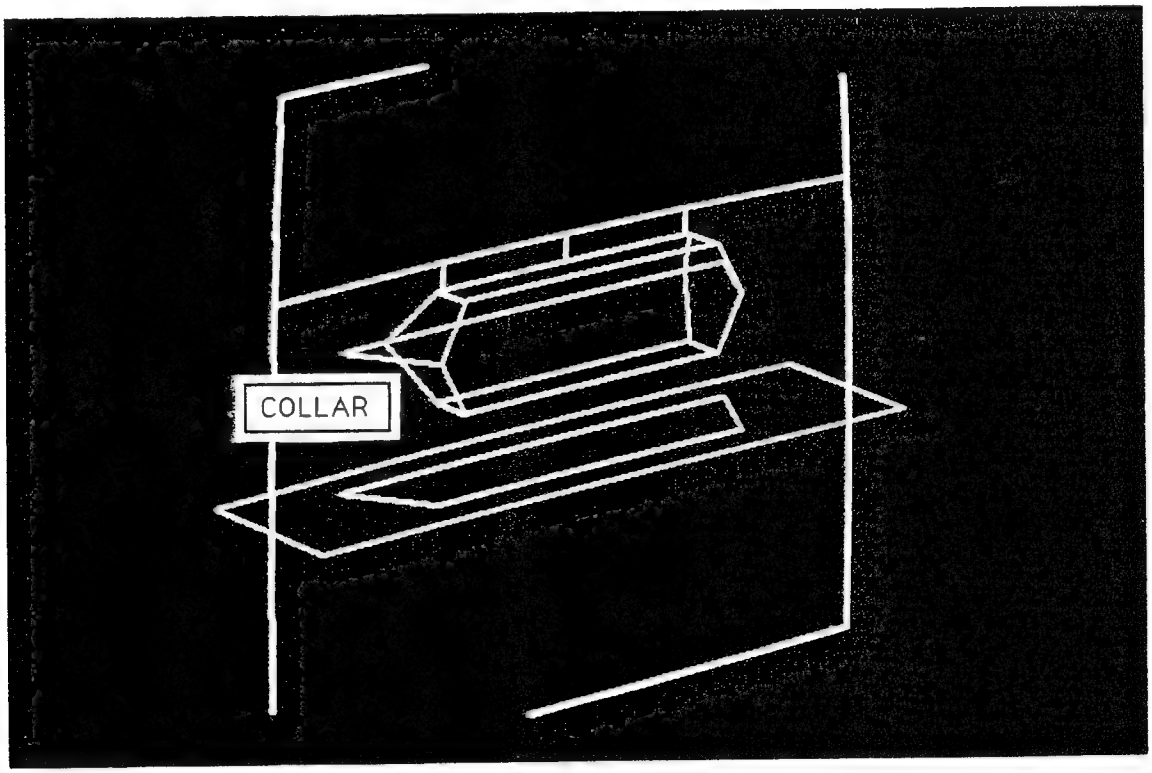
ROBOT	for the AdeptOne robot,
HAND	for the collar handling end-effector,
DESTCKER	for the destacker device,
TURNER	for the turning device,
PRESSER	for the pressing device,
PRTABLE	for the pressing table, and
COLLAR	for the apparel shirt collar.

Initial device variable and switch settings are specified by the operator in the manner prescribed in Appendix G along with information defining the geometry of the collars which are to be sourced to the workstation by means of the destacker. Figure 6.6(a) and 6.6(b) show two views of the AAW in simulation, either of which could be displayed on the monitor screen of the GS when ODCAP's planning programs (PLANNER.EXE in the UIP and SIMULATE.EXE in the GS) are executed following initialization. An additional ODCAP feature not discussed in Chapter IV is illustrated by Figure 6.6(b) which shows two COLLARs situated in the device DESTCKER. ODCAP permits an object to be simulated using different wire-frame models known as forms. Two such forms are depicted in Figure 6.6(b) for the object COLLAR, one form which consists of the object in a flattened state and another form with its upper and lower plies separated by gravity.

The operator with ODCAP's assistance creates the desired collar turning and pressing job plan one module at a time by selecting (in order) the five assembly tasks discussed in the previous section. The finished Master Plan produced for the AAW is listed in two parts in Figures 6.7 and 6.8. A **repeat##()():until_counter_done###()()** conditional module set (situated at E001 and E002) is placed around the five assembly action modules to cause repetition of the assembly job for multiple collars. The plan contains implicit modules (identical to the ones utilized in the



(a) Overall View



(b) DESTCKER View

Figure 6.6. AAW Simulation

```
I014  intro_dev_swt_info
      (PRESSER)
      (ndevswt:PRESSER:SS, devswti:PRESSER:SS)
I013  intro_dev_var_info
      (PRESSER)
      (ndevvar:PRESSER:SS, devvari:PRESSER:SS)
I010  intro_dev_swt_info
      (TURNER)
      (ndevswt:TURNER:SS, devswti:TURNER:SS)
I009  intro_dev_var_info
      (TURNER)
      (ndevvar:TURNER:SS, devvari:TURNER:SS)
I007  intro_dev_swt_info
      (DESTCKER)
      (ndevswt:DESTCKER:SS, devswti:DESTCKER:SS)
I006  intro_dev_var_info
      (DESTCKER)
      (ndevvar:DESTCKER:SS, devvari:DESTCKER:SS)
I005  intro_dev_swt_info
      (HAND)
      (ndevswt:HAND:SS, devswti:HAND:SS)
I004  intro_dev_var_info
      (HAND)
      (ndevvar:HAND:SS, devvari:HAND:SS)
I003  intro_dev_var_info
      (ROBOT)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS)
E001  repeat001
      ()
      (i001:SS, count001:SS)
E008  unload_device
      (DESTCKER, COLLAR, ROBOT, HAND,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       ndevvar:DESTCKER:SS, devvari:DESTCKER:SS,
       ndevswt:DESTCKER:SS, devswti:DESTCKER:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       ndevvar:DESTCKER:SS, devvari:DESTCKER:SS,
       ndevswt:DESTCKER:SS, devswti:DESTCKER:SS,
       nobsatc:HAND:SS, obsatci:HAND:SS)
```

Figure 6.7. AAW Master Plan Listing (Part I)

```
E011  load_and_activate_device
      (TURNER, COLLAR, ROBOT, HAND,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       nobsatc:HAND:SS, obsatci:HAND:SS,
       ndevvar:TURNER:SS, devvari:TURNER:SS,
       ndevswt:TURNER:SS, devswti:TURNER:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       nobsatc:HAND:SS, obsatci:HAND:SS,
       ndevvar:TURNER:SS, devvari:TURNER:SS,
       ndevswt:TURNER:SS, devswti:TURNER:SS)
E012  unload_device
      (TURNER, COLLAR, ROBOT, HAND,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       ndevvar:TURNER:SS, devvari:TURNER:SS,
       ndevswt:TURNER:SS, devswti:TURNER:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       ndevvar:TURNER:SS, devvari:TURNER:SS,
       ndevswt:TURNER:SS, devswti:TURNER:SS,
       nobsatc:HAND:SS, obsatci:HAND:SS)
E015  load_and_activate_device
      (PRESSER, COLLAR, ROBOT, HAND,
       ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       nobsatc:HAND:SS, obsatci:HAND:SS,
       ndevvar:PRESSER:SS, devvari:PRESSER:SS,
       ndevswt:PRESSER:SS, devswti:PRESSER:SS)
      (ndevvar:ROBOT:SS, devvari:ROBOT:SS,
       ndevvar:HAND:SS, devvari:HAND:SS,
       ndevswt:HAND:SS, devswti:HAND:SS,
       nobsatc:HAND:SS, obsatci:HAND:SS,
       ndevvar:PRESSER:SS, devvari:PRESSER:SS,
       ndevswt:PRESSER:SS, devswti:PRESSER:SS)
E016  press_obs
      (PRTABLE, PRESSER, COLLAR,
       ndevvar:PRESSER:SS, devvari:PRESSER:SS,
       ndevswt:PRESSER:SS, devswti:PRESSER:SS)
      (ndevvar:PRESSER:SS, devvari:PRESSER:SS,
       ndevswt:PRESSER:SS, devswti:PRESSER:SS)
E002  until_counter_done001
      (i001:SS, count001:SS)
      ()
```

Figure 6.8. AAW Master Plan Listing (Part II)

WIDGET assembly job plan) at the beginning of the plan which introduce variable-structures containing AAW device variable and switch information.

The action module **unload_device()()** is used twice (E008 and E012) to unload the destacker and turning devices by commanding the robot end-effector to acquire and remove one collar from the device specified in the module's FINITE-STATEs. The module requires input data-structures defining the status of all the devices involved in the module's execution. Position information for the collar, which is acquired by the end-effector, is not required for module input because the task involves end-effector motion relative to the device being unloaded. When unloading either the destacker or the turning device, the robot maneuvers the end-effector to a location, relative to these devices, where a collar is "believed" to be ready for acquisition. No sensing operations are performed by **unload_device()()** to verify that the collar is ready for unloading. Successful performance of the unloading task requires that the AAW destacker and turning devices manipulate the collar without error when performing their respective destacking and turning activities. Two different applications of the module **load_and_activate_device()()** (E011 and E015) carry out the main turning and pressing tasks. Collar position/orientation information is passed to the module in the variable-structures nobsatc:HAND:SS and obsatci:HAND:SS. The task uses the geometric relationship between the end-effector and

the grasped collar defined in these structures to maneuver the collar with the end-effector such that successful device loading is achieved.

The two ODCAP action modules **unload_device()()** and **load_and_activate_device()()** require the use of a new type of function which is included in the function lists corresponding to these modules. This new function type is termed "device specific" and is described through the use of an example. The following list of functions is reproduced from the file UNLDDEV.MOD ("UNLDDEV" is the shortened name used to indicate the module **unload_device()()**) which contains the listings (one for each AAW computer: SS, RC, and VC) of the module's composite functions.

```
  .           .
  .           .
  .           .
E0060  dev_spec_DESTCKER
       (DEVA)
       ()
E0061  activate_switch
       (DEVA,
        PICKSTAT, OFF)
       ()
E0062  end_dev_spec
       (DEVA)
       ()
E0063  dev_spec_TURNER
       (DEVA)
       ()
E0064  activate_switch
       (DEVC,
        TRNPTSTA, UNTRAP)
       ()
E0065  end_dev_spec
       (DEVA)
       ()
  .           .
  .           .
  .           .
```

The proceeding functions represent a partial listing of those functions connected with the AAW computer labeled SS. Two pairs of device specific functions are shown namely dev_spec_DESTCKER()() (E0060) and end_dev_spec()() (E0062) and dev_spec_TURNER()() (E0063) and end_dev_spec()() (E0065). Device specific functions are inserted in pairs and are used to identify module functions which only apply to a specific value of a FINITE-STATE. For this example, if the FINITE-STATE DEVA (indicating some device name) is specified as DESTCKER, then function E0061 is included in the resulting SS subplan; if DEVA is TURNER, function E0064 is included. Device specific functions are needed due to the inability to separate data from procedure for every process in a given task (in this particular case, data refers to the specific device selected for collar unloading). When ODCAP assembles the subplans at the conclusion of a planning session, it omits from the subplans those functions which are barred from inclusion by device specific functions. Device specific functions are never included in any subplan.

The conversion of the AAW subplans into actual programming code is performed automatically by ODCAP when the operator activates the program PROGEN.EXE in the UIP. This program incorporates all the actions carried out by the Planning Level's program generator (see Figure 3.10 on page 55). Program generation consists primarily of function

matching, in which an ODCAP function is exchanged for a "C" or "V+" function (depending on which subplan is undergoing conversion) which implements the corresponding process on the AAW hardware. Appendix I discusses program generation techniques using the AAW collar turning and pressing job plan for examples.

On-Line Operation

Once program generation is completed, the operator can use the resulting language code files in the same manner as the source code files developed without ODCAP. The operator must transfer (manually) each file to its corresponding computer and compile the code into an executable format using the compiler native to the workstation computer for which the program is intended (RC program files written in "V+" require no such compilation). Execution of planned workstation activities begins when these executable programs are activated by the operator as prescribed in Appendix G.

Position and orientation transformations associated with the kinematics of the AAW devices and the shirt collar must be determined prior to planning and placed in the appropriate files in the UIP. These transformations are derived from the static kinematic data (information which is only altered if device mechanisms are changed or if the devices are repositioned relative to the workstation's base coordinate frame) which can be determined from device specifications and by calibration procedures. If changes are

made to the static data and other related workstation data, an existent Master Plan must be "re-simulated" so that ODCAP can alter the plan's input data in the appropriate manner. Re-simulation procedures must also be conducted if changes are made to the initial device data.

The AAW is not optimally suited for demonstrating the utility of the ODCAP software package because of its inherent inflexibility. The AAW devices and the task sequence associated with the utilization of these devices are essentially fixed and only address one type of shirt collar at present. Yet even with a workstation such as the AAW, ODCAP provides a user-friendly, modular environment for planning, replanning, and simulating workstation activities. The AAW control computers known as SS and VC were used to implement the ODCAP software for AAW job planning with the SS and VC acting as the UIP and GS respectively. The performance of the collar turning and pressing tasks using the ODCAP-generated control programs was practically identical to that of the programs developed without ODCAP. Demonstrations of the actual and simulated operation of the AAW is available on videotape from the Mechanical Engineering Department, Clemson University.

CHAPTER VII

CONCLUSIONS AND RECOMMENDATIONS

Conclusions

A hierarchical control and planning scheme has been devised which combines man-machine interface with modular concepts to make the programming and reprogramming of PC-controlled workstations fast and flexible. The complexity and difficulty of creating control programs for workstation tasks is often overlooked for more conventional control issues. Currently, no consensus exists in hierarchical control theory regarding the definition of hierarchical control structures. Control hierarchies are usually represented as multi-level block diagrams with information passed between levels in the form of control signals. Researchers agree that levels at the top of the hierarchy should possess a greater degree of abstraction and intelligence than levels at the bottom, but they do not indicate how these distinctions can be realized in actual control structures.

This research effort uses hierarchical control theory as a foundation for developing software-based control structures. These structures, when activated, direct the physical performance of workstation activities. The results show that a three-level hierarchy is suited to the organization of workstation activities with the lowest-level activity (process) corresponding to a single device action. The

middle-level, known as the task-level, serves as the level at which a workstation operator interacts with the planning scheme. An operator who is familiar with workstation operation on a task-level becomes an effective planner if he/she is provided with adequate support. The highest hierarchical level consists of a complete program entity known as a plan which is not additive; different plans must be individually developed for different workstation jobs.

A three-tiered control and planning scheme is established to interface with a workstation operator and to manipulate the program structures which correspond to the activity hierarchy so that useful control programs are created. Software is shown to be the instrument of workstation control whereas device hardware represents the means of implementation. A significant contribution of the research is the differentiation of software and hardware issues in the control and planning scheme. The scheme's three tiers are specified as the Planning, Coordination, and Hardware Levels. The creation of workstation control programs is carried out at the Planning Level while the Coordination and Hardware Levels correspond to the execution of these programs by workstation computing resources and to the performance of planned activities by workstation hardware respectively.

A workstation operator interfacing with the scheme at the Planning Level performs the duties of deciding which tasks are required for a certain job and how these tasks

should be sequenced. Programming workstation computers typically demands that an operator perform many more difficult duties than task selection and sequencing. These duties may require information not readily available to the operator and programming skills which he/she does not possess. In addition to reducing the duties associated with job planning, the hierarchical scheme provides several systems to assist the operator with task selection. One system, which is used in some manner by most off-line planning schemes, implements graphics simulation of the workstation performing operator-selected tasks. Extensive use of this system has shown that having the operator view a task's effect on workstation apparatus is beneficial to his/her decision-making capability.

The input/output data connected to a hierarchical program structure is considered a separate entity from the structure's procedures on both a task-level and a process-level. The research has demonstrated that this separation is fundamental to the identification of hardware-related information. The exchange of information between task and process levels is carried out by special data conversion functions. Data structures are developed to facilitate the access of task information from resource-linked data files. Grouping task information based on workstation resources, such as devices and objects, is advantageous because it permits such information to be altered or expanded independent of procedural changes. Every job plan is divided into

two distinct parts, an introductory data section and a module sequence. The first part contains the data which is manipulated by the procedural program structures (modules) listed in the second part. Integration of plan data to plan procedures is achieved autonomously by the control and planning scheme.

Two Planning Level systems make use of data/procedure separation to assist the operator in deciding what tasks are needed in a job. Task selection is based on feasibility and the satisfaction of job goals. Rule-base systems are developed to evaluate these considerations and to determine which tasks apply to a given job. Workstation information is transformed from quantitative data to qualitative facts; these facts are asserted to the various data bases which supply the rule-bases. The use of rule-base systems in this manner enables the creator of the rules to set planning objectives without having to analyze directly the data which characterizes workstation operation.

Examination of the tasks contained in typical jobs has revealed that a workstation operator performing job planning does not require explicit knowledge about every task. Some computational tasks perform activities such as data initialization or system communication. Such tasks are termed implicit and can be added to jobs without the operator's consent. Most tasks, however, affect workstation operation such that the operator should take part in deciding when they should be inserted into a job's task sequence. An

Explicit Rule-Base (ERB) system and an Implicit Rule-Base (IRB) system have been devised for selecting tasks with and without the workstation operator's knowledge respectively.

Facts generated for use by the ERB system relate primarily to workstation and job plan status. The ERB system produces recommendations which convey to the operator the various task alternatives. The results show that the final selection of one of these tasks is a duty which should be performed by the operator. Task alternatives may be few in number, yet in most cases, there exists no clear indicator for selecting one task over another. The workstation operator possesses the intuitive skills needed for task selection which cannot be easily duplicated using computer-based AI techniques.

The IRB system operates autonomously, deciding how tasks receive input data in the form of data structures. The system can unilaterally reject an operator-selected task if it is unable to identify implicit tasks which produce data structures that satisfy the input data requirements of the selected task. The workstation operator is not burdened with these programming details since he/she is not interested in how tasks receive data but in how they contribute to the realization of planning objectives.

One drawback associated with hierarchical schemes is that they result in larger, more complex control programs than those produced by direct system programming. This

increase in program overhead can be attributed to the modularity corresponding to the hierarchical program structures. The benefits attained by dividing workstation planning duties between an operator and system programmers offset the costs related to programming complexity. The workstation operator can focus completely on those task-level issues which affect workstation behavior. System programmers are required to create the program structures which correspond to tasks and processes, but they are freed from considerations such as task sequencing and the integration of resource-linked data to plan procedures.

The proposed hierarchical workstation control and planning scheme is implemented in the C/Prolog software package called ODCAP (Operator-Driven Controller And Planner). ODCAP has been applied to a hypothetical workstation which performs robot-assisted assembly tasks. Planning demonstrations have been conducted which verify the scheme's usefulness to workstation programming. A menu-driven interface is combined with the graphics simulation to provide the workstation operator a user-friendly environment for interacting with ODCAP. The "replanning" of a task sequence is achieved by allowing the operator to insert additional tasks into the sequence. The results show that replanning is an important feature because, in many cases, the tasks originally included in a job cannot handle every unforeseen contingency which may arise when the job is performed.

ODCAP is also applied to a real-world manufacturing workstation (the AAW) which automates the turning and pressing operations connected with apparel shirt collar manufacture. The real-time control of the AAW is carried out by PC-type computers; the configuration of AAW resources (including the control computers) conforms to one acceptable to ODCAP. Process-level subplans are converted into language code control programs by one of ODCAP's Planning Level systems termed the program generator. The successful use of these programs for implementing on-line AAW activities proves that the control and planning scheme can be applied to real-world problems.

The limitations of ODCAP-generated control programs coincide with those of any software intended for use on a real-time system controlled by PC-type computers. Such computers are usually capable of controlling only one task at a time. The monitoring of output from sensing devices using techniques such as interrupts or polling can only be conducted by the functions within a single module. No provision has been made for implementing continuous device monitoring during the performance of a job. Control strategies such as actuator/sensor feedback loops can be directed by program functions if these functions can be executed concurrently with subplan functions.

Recommendations

The recommendations address two sets of issues, relating to the improvement of the ODCAP software which implements the hierarchical semi-autonomous workstation control and planning scheme and to the consideration of fundamental research concepts. Extensive usage of ODCAP for job planning has uncovered areas where the software could be improved and expanded. The suggested improvements could be implemented without radically altering ODCAP's current disposition. Research issues such as the scheme's integration with other control methods and the use of hierarchical control with hardware configurations other than workstations are examined.

Implementation Issues

The current version of ODCAP has no provision for allowing the workstation operator to review the tasks which he/she previously selected, tasks whose corresponding program modules comprise the existent job plan. The operator is presented with textual descriptions for task recommendations and with graphics simulation of the workstation performing selected tasks. The user interface should provide useful information on the status of the job plan and the condition of workstation resources. These items change as tasks are added to a job; keeping the operator informed of the changes can benefit his/her decision-making regarding task selection and sequencing. The rules for ODCAP's two

rule-base systems are presently developed by a system programmer using the Prolog programming language. Since rules define planning objectives and task feasibility, the workstation operator should be involved in their construction. Algorithms are needed which convert operator specifications relating to planning objectives into computer coded rules and facts.

The replacement of the wire-frame modeling currently used by ODCAP graphics to solid modeling would give the operator a more realistic conception of the workstation in action. Error checking should be carried out during a planning session and after plan creation. Presently, ODCAP has no mechanism for error checking other than forbidding the inclusion of tasks for which input data cannot be determined. Additional systems could be created for the Planning Level which monitor task performance issues such as obstacle avoidance and the enforcement of time and force constraints. The ODCAP planning algorithm does not permit the operator to delete tasks from an existing job nor does it allow him/her to alter the execution sequence of the modules comprising a job plan (except with the addition of conditional modules to the plan). The power and sophistication of the ODCAP software would be enhanced if these features could be implemented on a Planning Level. Job planning experiments have demonstrated that ODCAP is nearing the operational limits imposed by the PC/ATs which implement the package. ODCAP's adaptation to more powerful computing resources would

decrease the development time connected with job planning and would allow the addition of the features discussed in this section.

Fundamental Issues

Future research should be conducted on the organization of system activities as a hierarchy and should develop precise methods for identifying how activities should be distributed among various hierarchical levels. Heuristic approaches were utilized in this research which define a task as an activity whose performance affects the status of workstation objects in some prescribed manner. The activities below a task-level (processes) were made to correspond to elemental machine operations.

The issue of data/procedure separation is interrelated to the construction of the activity hierarchy. The grouping of information into data structures at each hierarchical level must be carried out such that the information remains compatible with the procedural elements residing at each level. The duty of integrating data to plan procedures should be removed from the human operator wherever possible and should be given to autonomous systems which can obtain needed data from resource-linked data files. The integration of manufacturing information to the performance and design characteristics of workstation devices and objects is perceived as the end result of further research into

data/procedure separation at a workstation level. Manufacturing information can be obtained either from CAD-based sources or from operator-interactive teach methods involving actual workstation hardware.

The control and planning scheme developed in this dissertation adapted rule-base systems to assist with the autonomous and semi-autonomous selection of tasks for inclusion into workstation jobs. Other control theories such as neural networks, adaptive control, and fuzzy logic control could be applied to perform the duties of these rule-base systems. Such applications, however, would require that workstation information be arranged into forms suitable to the needs of the particular control theory.

The planning scheme should be extended to enable task concurrency. Tasks which utilize different workstation resources and do not interfere with one another can be performed simultaneously. Planning Level systems could assess the feasibility of task concurrency and impose it automatically even if the operator selects the tasks during separate planning steps. The amount of task concurrency would be restricted by the number of computing agents contained within the workstation control computers.

The operator-assisted planning techniques developed at a task-level can theoretically be applied to any hierarchical level. For instance, a similar planning scheme can be created for a system programmer which helps him/her in determining the process sequence of a single task. The

scheme can also be shifted vertically to levels above workstation control which concern shop-wide or factory-wide issues. Hierarchical control schemes have been implemented at a factory level but they do not use operator-assisted planning to determine activity sequences. The elimination of human participation in activity planning is not regarded as a desired or realistic goal. Alternative research should focus on providing human planners with "tools" which reduce the burdensome details associated with planning.

APPENDICES

Appendix A

Conditional Modules

A conditional module is the only module type (i.e. conditional, computational, and action) which can alter the execution sequence of tasks associated with a job plan. When a task corresponding to either an action or a computational module is completed, the next task which is performed is the task corresponding to the module listed below the previously executed module in the Master Plan. Conditional modules can change the execution sequence to one different from the module listing of the Master Plan. Conditional modules are developed for the proposed planning scheme which operate similarly to the conditional programming structures used by the language VAL II [71].

Two kinds of conditional modules are considered, those which effect "looping" in a module execution sequence and those which cause "branching". Looping modules force other plan modules to undergo repeated execution. A "conditional test" is performed by computational functions within the looping modules to determine when module execution repetition should cease. The conditional test can be based on any workstation-related information. Branching modules can direct that certain modules be executed or bypassed dependent on the evaluation of the branching module's conditional test. The proposed planning scheme uses two varieties of

looping conditionals and four varieties of branching conditionals. These conditionals have been developed into modules on a task-level and functions on a process-level. Only the task-level conditionals are examined in this appendix. A table placed in Figure A.1 lists the six conditional varieties adopted for the scheme; the first two varieties, REPEAT-UNTIL and WHILE-ENDWHILE, denote looping conditionals while the remaining varieties represent branching conditionals.

Conditional modules (and functions) always come in sets of more than one module (function). The modules within a given set enclose groups of plan modules (may be just one module) whose execution depends on the results of the conditional test associated with the conditional modules. The table in Figure A.1 specifies the number of modules contained in each conditional set for the six conditional varieties. Different module sets can be developed (by a system programmer) for each of the six varieties. A method has been adopted for naming conditional module sets are named based on their variety. The shortened versions (eight or less alphanumeric characters) of these names are included in the conditional set table with the "?" representing some alphanumeric character identifying a particular conditional module set. The "#" character in the varieties, IF-ELSEIF-...-ENDIF and IF-ELSEIF-...-ELSE-ENDIF, signifies the variable number of plan module groups which the (# + 1) modules within each set can enclose.

Conditional Varieties	Number of Modules in Set	Module Set Name (Shortened Version)
REPEAT-UNTIL	2	RPT?????
WHILE-ENDWHILE	2	WHL?????
IF-ENDIF	2	IF1?????
IF-ELSE-ENDIF	3	IF2?????
IF-ELSEIF-...-ENDIF	(# + 1)	IFE#????
IF-ELSEIF-...-ELSE-ENDIF	(# + 1)	IFS#????

Figure A.1. Conditional Set Table

Two conditional module set examples are provided for both looping and branching conditionals to illustrate their operation. The first module set example is labeled as **repeat###()():until_counter_done###()()** where the characters "###" denote a three-digit conditional set identification number which is specified when the module set is included into a Master Plan. These identification numbers assist in differentiating one conditional module set from another. The colon in the module set label divides the two modules, **repeat###()()** and **until_counter_done###()()**, which comprise this set. The shortened name for this looping conditional is RPTCOUNT where "RPT" indicates that the module set is of variety REPEAT-UNTIL and "COUNT" represents the particular label that the module set **repeat###()(): until_counter_done###()()** is provided.

Operation of the RPTCOUNT conditional set is shown by describing the set's effect on the execution sequence of some plan. The RPTCOUNT set may be included into a plan such that a portion of the Master Plan resembles the following.

```
        .
        .
        .
repeat001
()
(i001:SS, count001:SS)

{Module Group A}

until_counter_done001
(i001:SS, count001:SS)
()
```

{Module Group B}
.
.
.

RPTCOUNT implements a fixed number of module execution repetitions; the value for this number is specified by the operator when the module set is selected during planning. The conditional test which determines when the operator-specified number of repetitions is completed is performed by the **until_counter_done001()()** module. (Variable-structures within conditional modules can be "tagged" with the conditional set's identification number as indicated by i001:SS and count001:SS.)

Execution of the **repeat001()()** module would be followed by the sequential execution of every plan module within Module Group A. The module **until_counter_done001()()** would subsequently determine if the modules in Group A should be executed again by comparing the number of repetitions previously performed (a value stored in i001:SS) with the total number specified by the operator (stored in count001:SS). If all the repetitions of Module Group A have been carried out then plan execution will continue with the first module in Module Group B.

The next example of a looping conditional is the module set **while_object_on_device###()():endwhile###()()** with the shortened name of WHLOBDEV. The primary difference between the conditional varieties, REPEAT-UNTIL (as represented by RPTCOUNT) and WHILE-ENDWHILE (as represented by WHLOBDEV),

is that the conditional test associated with REPEAT-UNTIL conditionals is performed by the second module in the conditional's set whereas the test for WHILE-ENDWHILE conditionals is included in the first module in the set. The WHLOBDEV module set alters the module execution sequence based on whether or not some object is located on some device. WHLOBDEV may be incorporated into a plan in the following manner.

```
          .
          .
          .
     while_object_on_device002
     (BLOCK, TABLE, nobsatc:TABLE:SS,
      obsatci:TABLE:SS)
     (nobsatc:TABLE:SS, obsatci:TABLE:SS)

    {Module Group A}

     endwhile002
     (BLOCK, TABLE)
     ()

    {Module Group B}
          .
          .
          .
```

This conditional example illustrates the fact that every module within any conditional set always has the same FINITE-STATEs included in its input data. The variable-structures nobsatc:TABLE:SS and obsatci:TABLE:SS hold information related to all the objects currently attached to the device TABLE. The module **while_object_on_device###()()** uses this data to determine if at least one object BLOCK is presently attached to TABLE. The execution sequence will continue with Module Group A if one (or more) BLOCKs is

affixed to TABLE; otherwise, the sequence will move to the first module in Group B. After the last module in Group A is executed (assuming at least one BLOCK was attached to TABLE), the conditional test associated with the module **while_object_on_device###()()** is repeated to determine if the modules in Group A should be executed a second time.

The third example concerns the variety IF-ELSE-ENDIF and involves the three module set **if_device_ready###()(): else###()():endif###()()** with the shortened name IF2DVRDY. The operation of IF2DVRDY is demonstrated with the following Master Plan excerpt.

```
        .
        .
        .
 if_device_ready003
 (ROBOT, ndevvar:ROBOT:SS,
  devvari:ROBOT:SS)
 (ndevvar:ROBOT:SS, devvari:ROBOT:SS)

{Module Group A}

 else003
 (ROBOT)
 ()

{Module Group B}

 endif003
 (ROBOT)
 ()

{Module Group C}
        .
        .
      .
        .
```

The first module in this set performs a conditional test which evaluates the information in variable-structures ndevvar:ROBOT:SS and devvari:ROBOT:SS. The end result of

this evaluation leads to the execution of the modules in either Module Group A or Module Group B. When the execution of all the modules in Groups A or B is completed, the execution sequence continues with the first module in Group C.

The final example involves a branching conditional of the variety IF-ELSEIF-...-ENDIF which implements branching of the execution sequence with multiple conditional tests. The name given to the set is **if_object_drilled###()():elseif_milled###()():elseif_stamped###()():endif###()()** with the shortened name IFE3MACH where the "3" contained in the name indicates the number of module groups enclosed by the set's modules. The following module listing illustrates the operation of IFE3MACH.

```
          .
          .
          .
 if_object_drilled004
 (PART, prop:PART:VC)
 (prop:PART:VC)

{Module Group A}

 elseif_milled004
 (PART, prop:PART:VC)
 (prop:PART:VC)

{Module Group B}

 elseif_stamped004
 (PART, prop:PART:VC)
 (prop:PART:VC)

{Module Group C}

 endif004
 (PART)
 ()
```

{Module Group D}

.
.
.

The first three modules in set IFE3MACH each implement one conditional test using the data in variable-structure prop: PART:VC which defines the various properties of the object PART. The conditional tests are performed separately, from top to bottom, until one is evaluated as true which subsequently causes the modules within the appropriate group to be executed. If an object labeled PART, for example, were "milled" and "stamped" but not "drilled", the conditional test for **if_object_drilled004()()** would fail. The next test corresponding to the **elseif_milled004()()** module would succeed forcing the modules in Group B to be executed. The conditional test in **elseif_stamped004()()** would not be attempted even though the object is "stamped". Once the execution of Module Group B is completed, the execution sequence will continue to Module Group D. If an object PART does not possess any of the three aforementioned properties, the modules in Groups A and B and C will not be executed.

Appendix B

Task/Module Example

An example of a module which executes a robotic task is reviewed to provide insight on how process-level program structures, termed functions, are organized into modules. The operation and purpose of many of these functions are described individually, revealing the manner by which the module executes the desired task. The function listings pertaining to the task's process sequences are included in the appendix. Every module incorporated into a workstation job plan must have function listings similar to the ones provided in this example. The listings are created by a programmer familiar with process-level operations and are placed into data files. The planning scheme accesses a module's function listings when assembling job subplans.

The robotic task presented in this appendix involves a pick-and-place activity for a single object. Takanashi et al. [72] defines three assembly operations for robots manipulating objects as follows: (1) handling, (2) transfer, and (3) joining. The pick-and-place example task involves two of these operations, namely handling and transfer. The module which performs the task is **pick_place_obs_to_dev()()**

and is specified with its input and output data in the following form.

```
pick_and_place_obs_to_dev
(OBSA, DEVA, DEVB, DEVC,
 DEVD, DEVE, ndevvar:DEVA:SS,
 devvari:DEVA:SS, nobsatc:DEVD:SS,
 obsatci:DEVD:SS, nobsatc:DEVE:SS,
 obsatci:DEVE:SS)
(ndevvar:DEVA:SS, devvari:DEVA:SS,
 nobsatc:DEVD:SS, obsatci:DEVD:SS,
 nobsatc:DEVE:SS, obsatci:DEVE:SS)
```

The module requires specification of six FINITE-STATEs representing the devices and object which are affected by task execution. The generic labels for these FINITE-STATEs are OBSA, DEVA, DEVB, DEVC, DEVD, and DEVE signifying the one object and five devices, respectively, associated with the task. Specific values for these FINITE-STATEs are determined during planning and are substituted for the generic labels when the module is inserted into a job plan.

The pick-and-place task carried out by the action module **pick_place_obs_to_dev()()** involves the acquisition, transport, and release of an object OBSA by a gripping device DEVC. The gripping device is attached to a coupling device DEVB which, in turn, is connected to the wrist of a robot device DEVA. The device DEVD represents the device from which OBSA is acquired while DEVE is the device to which the object is placed. Different FINITE-STATE values can be chosen for **pick_place_obs_to_dev()()**, implying that the same task/module will work with any suitable combination of devices and object.

The module contains six identical variable-structures in its input data list and in its output data list. Two of these structures, ndevvar:DEVA:SS and devvari:DEVA:SS, provide information on the joint configuration of the robot device DEVA. These variable-structures are included in the output because the robot's configuration changes after completing the pick-and-place task. The data incorporated in the four variable-structures: nobsatc:DEVD:SS, obsatci: DEVD:SS, nobsatc:DEVE:SS, and obsatci:DEVE:SS, defines the position and orientation of all the objects attached to the respective devices DEVD and DEVE. The module utilizes this information when determining the robot trajectories required to acquire and transport the object OBSA. The variable-structures nobsatc:DEVD:SS and obsatci:DEVD:SS are altered by the removal of position/orientation data for one OBSA object following the completion of the pick-and-place task, due to the fact that the object has been moved to DEVE. Consequently, the data structures associated with DEVE are augmented by the position/orientation data of the transported object. Modules (and functions) should, in general, include the information in their input data which is needed for task execution and should output those data structures which are affected by task execution.

The function listings for **pick_place_obs_to_dev()()** are shown in eight consecutive parts in Figures B.1 through B.8 with the functions numbered from "E0001" to "E0076". The

FCT #	VC LISTING	SS LISTING	RC LISTING
E0001		enter_modpart (PPOBSDEV, 0000, ndevvarDEVA, devvariDEVA, nobsatcDEVD, obsatciDEVD, nobsatcDEVE, obsatciDEVE) (nrbtjtcoords, rbtjtcoords, nobsatcdevd, obsatcidevd, nobsatcdeve, obsatcideve)	enter_modpart (PPOBSDEV, 0000) ()
E0002		copy_devvarinfo (nrbtjtcoords, rbtjtcoords) (nrbtjtcoords, rbtjtcoords, noldrbtjtcds, oldrbtjtcds)	
E0003		copy_obsatcinfo (nobsatcdevd, obsatcidevd) (nobsatcdevd, obsatcidevd, oldnobsatcdevd, oldobsatcidevd)	
E0004		get_obstrans (OBSA, nobsatcdevd, obsatcidevd) (nobsatcdevd, obsatcidevd, obstrans)	
E0005		copy_obsatcinfo (nobsatcdeve, obsatcideve) (nobsatcdeve, obsatcideve, oldnobsatcdeve, oldobsatcideve)	
E0006		intro_sgdoub () (appdist)	
E0007		intro_app_info (OBSA, DEVC) (apptransobs, apptransdev)	

Figure B.1. Function Listings for Module Example (Part I)

FCT #	VC LISTING	SS LISTING	RC LISTING
E0008		approach_obs (OBSA, DEVC, obstrans, apptransobs, apptransdev, appdist) (pathpt1, pathpt2, obstrans)	
E0009		intro_tooltrans (DEVA, DEVB, DEVC) (tooltrans)	
E0010		move_tool_toward (DEVC, DEVA, pathpt1, pathpt2, tooltrans, nrbtjtcoords, rbtjtcoords) (traj1, linetraj2, nrbtjtcoords, rbtjtcoords, resp12)	
E0011		intro_sgdoub () (depdist)	
E0012		intro_dvftrans (DEVD) (devtrans)	
E0013		intro_ddvob_info (OBSA, DEVD, obstrans, devtrans) (dvobtransdev, dvobtransobs, obstrans, devtrans)	

Figure B.2. Function Listings for Module Example (Part II)

FCT #	VC LISTING	SS LISTING	RC LISTING
E0014		dep_dev_w_obs (OBSA, DEVC, DEVD, devtrans, dvobtransdev, dvobtransobs, depdist) (pathpt1)	
E0015		intro_app_info (OBSA, DEVC) (apptransobs, apptransdev)	
E0016		det_obsdevtrans (OBSA, DEVC, apptransobs, apptransdev) (obsdevtrans)	
E0017		intro_tooltrans (DEVA, DEVB, DEVC) (tooltrans)	
E0018		move_obs_away (DEVC, DEVA, pathpt1, tooltrans, nrbtjtcoords, rbtjtcoords, obsdevtrans) (linetraj4, nrbtjtcoords, rbtjtcoords, obsdevtrans, obstrans, resp4)	
E0019		intro_sgdoub () (appdist)	
E0020		intro_dvftrans (DEVE) (devtrans)	
E0021		intro_advob_info (OBSA, DEVE) (dvobtransdev, dvobtransobs)	

Figure B.3. Function Listings for Module Example (Part III)

FCT #	VC LISTING	SS LISTING	RC LISTING
E0022		app_dev_w_obs (OBSA, DEVC, DEVE, devtrans, dvobtransdev, dvobtransobs, appdist) (pathpt1, pathpt2)	
E0023		intro_tooltrans (DEVA, DEVB, DEVC) (tooltrans)	
E0024		move_obs_toward (DEVC, DEVA, pathpt1, pathpt2, tooltrans, nrbtjtcoords, rbtjtcoords, obsdevtrans) (traj5, linetraj6, nrbtjtcoords, rbtjtcoords, obsdevtrans, obstrans, resp56)	
E0025		intro_sgdoub () (depdist)	
E0026		intro_app_info (OBSA, DEVC) (apptransobs, apptransdev)	
E0027		depart_obs (OBSA, DEVC, obstrans, apptransobs, apptransdev, depdist) (pathpt1, obstrans)	

Figure B.4. Function Listings for Module Example (Part IV)

FCT #	VC LISTING	SS LISTING	RC LISTING
E0028		intro_tooltrans (DEVA, DEVB, DEVC) (tooltrans)	
E0029		move_tool_away (DEVC, DEVA, pathpt1, tooltrans, nrbtjtcoords, rbtjtcoords) (linetraj8, nrbtjtcoords, rbtjtcoords, resp8)	
E0030		add_obstrans (OBSA, obstrans, nobsatcdeve, obsatcideve) (nobsatcdeve, obsatcideve)	
E0031		and_four_resp (resp12, resp4, resp56, resp8) (resp001)	
E0032		asm_mssge_RESP0 (resp001) (mssge, resp001)	
E0033		xmit_mssge_ss_rc (mssge) ()	recv_mssge_ss_rc () (mssge)
E0034			dis_mssge_RESP0 (mssge) (resp001)
E0035		if001 (resp001) ()	if001 (resp001) ()
E0036		asm_mssge_ADOU3 (traj1, linetraj2, 4) (mssge)	
E0037		xmit_mssge_ss_rc (mssge) ()	recv_mssge_ss_rc () (mssge)
E0038			dis_mssge_ADOU3 (mssge, 4) (traj, linetraj)

Figure B.5. Function Listings for Module Example (Part V)

FCT #	VC LISTING	SS LISTING	RC LISTING
E0039			wait_robot_done () ()
E0040			intro_sginte () (speed)
E0041			move_robot_jtintrp (DEVA, traj, speed) ()
E0042			wait_robot_done () ()
E0043			intro_sginte () (speed)
E0044			move_robot_strline (DEVA, linetraj, speed) ()
E0045			wait_robot_done () ()
E0046		break () ()	break () ()
E0047		activate_tool (DEVC, GRPSTATE, CLOSE) ()	
E0048		break () ()	break () ()
E0049		asm_mssge_ADOU4 (linetraj4, 4) (mssge)	
E0050		xmit_mssge_ss_rc (mssge) ()	recv_mssge_ss_rc () (mssge)
E0051			dis_mssge_ADOU4 (mssge, 4) (linetraj)
E0052			intro_sginte () (speed)
E0053			move_robot_strline (DEVA, linetraj, speed) ()

Figure B.6. Function Listings for Module Example (Part VI)

PCT #	VC LISTING	SS LISTING	RC LISTING
E0054		asm_mssge_ADOU3 (traj5, linetraj6, 4) (mssge)	wait_robot_done () ()
E0055		xmit_mssge_ss_rc (mssge) ()	recv_mssge_ss_rc () (mssge)
E0056			dis_mssge_ADOU3 (mssge, 4) (traj, linetraj)
E0057			intro_sginte () (speed)
E0058			move_robot_jtintrp (DEVA, traj, speed) ()
E0059			wait_robot_done () ()
E0060			intro_sginte () (speed)
E0061			move_robot_strline (DEVA, linetraj, speed) ()
E0062			wait_robot_done () ()
E0063		break () ()	break () ()
E0064		activate_tool (DEVC, GRPSTATE, OPEN) ()	
E0065		break () ()	break () ()
E0066		asm_mssge_ADOU4 (linetraj8, 4) (mssge)	
E0067		xmit_mssge_ss_rc (mssge) ()	recv_mssge_ss_rc () (mssge)
E0068			dis_mssge_ADOU4 (mssge, 4) (linetraj)

Figure B.7. Function Listings for Module Example (Part VII)

FCT #	VC LISTING	SS LISTING	RC LISTING
E0069			intro_sginte () (speed)
E0070			move_robot_strline (DEVA, linetraj, speed) ()
E0071		else001 () ()	else001 () ()
E0072		copy_devvarinfo (noldrbtjtcds, oldrbtjtcds) (noldrbtjtcds, oldrbtjtcds, nrbtjtcoords, rbtjtcoords)	
E0073		copy_obsatcinfo (oldnobsatcdevd, oldobsatcidevd) (oldnobsatcdevd, oldobsatcidevd, nobsatcdevd, obsatcidevd)	
E0074		copy_obsatcinfo (oldnobsatcdeve, oldobsatcideve) (oldnobsatcdeve, oldobsatcideve, nobsatcdeve, obsatcideve)	
E0075		endif001 () ()	endif001 () ()
E0076		exit_modpart (PPOBSDEV, 0000, nrbtjtcoords, rbtjtcoords, nobsatcdevd, obsatcidevd, nobsatcdeve, obsatcideve) (ndevvarDEVA, devvariDEVA, nobsatcDEVD, obsatciDEVD, nobsatcDEVE, obsatciDEVE)	exit_modpart (PPOBSDEV, 0000) ()

Figure B.8. Function Listings for Module Example (Part VIII)

workstation which implements the pick-and-place task is controlled by three computers labeled the System Supervisor (SS), the Robot Controller (RC), and the Vision Controller (VC). The functions are listed in separate columns (as indicated by the headings at the top of each figure) corresponding to the individual process sequences which each workstation computer must perform for task execution. The **pick_place_obs_to_dev()()** module consists of functions in only two workstation control computers, the SS and the RC. The module's SS functions determine the motion trajectories of the robot device DEVA needed to achieve the object acquisition and transport activities. Communication functions, in turn, pass this information to the RC where action functions actuate the robot in the appropriate manner. The first column of the function listings in Figures B.1 through B.8 is left empty, indicating that the pick-and-place task involves no vision processes. The generic FINITE-STATE labels: OBSA, DEVA, DEVB, DEVC, DEVD, and DEVE would be replaced with specific FINITE-STATE values when the functions are incorporated into process-level subplans.

The SS and RC function listings begin with the function enter_modpart()() (E0001), a function (included in at least one function listing for every module) which converts task-level input data into process-level data. The action module **pick_place_obs_to_dev()()** has six input variable-structures corresponding to data held by the SS computer. Each of these structures is transformed into process-level data

structures by enter_modpart()(). For example, the information held in devvari:DEVA:SS is transferred to the process-level structure "rbtjtcoords". No data structures require conversion for the RC since all input data is connected to the SS computer. Only function listings which are completely devoid of any functions (such as the VC listing for **pick_place_obs_to_dev()()**) can omit enter_modpart()().

The SS function get_obstrans()() (E0004) produces a single variable-structure "obstrans" from the input data structures "nobsatcdevd" and "obsatcidevd" which define the position and orientation of all objects attached to DEVD. The data in "obstrans" represents a numerical transformation defining the geometric relationship between a base coordinate system (or frame) and a coordinate system affixed to the object OBSA. The base coordinate frame is termed the Workstation Base Frame; the position and orientation of objects and devices are considered "known" if their coordinate frames can be linked to the Workstation Base Frame either by a single transformation (such as that represented by "obstrans" for OBSA) or by a series of known transformations.

Determination of robot joint coordinates is achieved in **pick_place_obs_to_dev()()** by determining a series of transformations which define a geometric relationship between a coordinate frame attached to the robot's wrist and the Workstation Base Frame. Inverse kinematics is then used to compute the joint coordinates necessary to achieve the

desired wrist positions and orientations (see Paul [65] for information on numerical transformations and inverse kinematics). The position/orientation of the robot wrist is determined at critical locations along the path which the robot must traverse to achieve the desired pick-and-place task. The means by which the functions in the module **pick_place_obs_to_dev()()** obtain some of these positions/orientations is discussed.

The computational functions intro_sgdoub()() (E0006) and intro_app_info()() (E0007) (see the SS listing in Figure B.1) provide examples of functions which bring in data from the introductory data section. The introductory data section is formed during planning, containing information derived from the operator and from data files. The output variable-structure "appdist" in intro_sgdoub()() consists of a single data value which represents the distance that the gripping device DEVC (attached to the robot wrist) should be positioned away from OBSA before the gripping device is translated for object grasping. The intro_app_info()() function outputs two transformations represented by "apptransobs" and "apptransdev". These transformations determine the kinematics related to the movement of DEVC by the robot DEVA when approaching OBSA, prior to object acquisition.

Figure B.9 depicts the transformations and coordinate frames associated with the grasping motion. Transformations are indicated in the figure by arcs which connect two

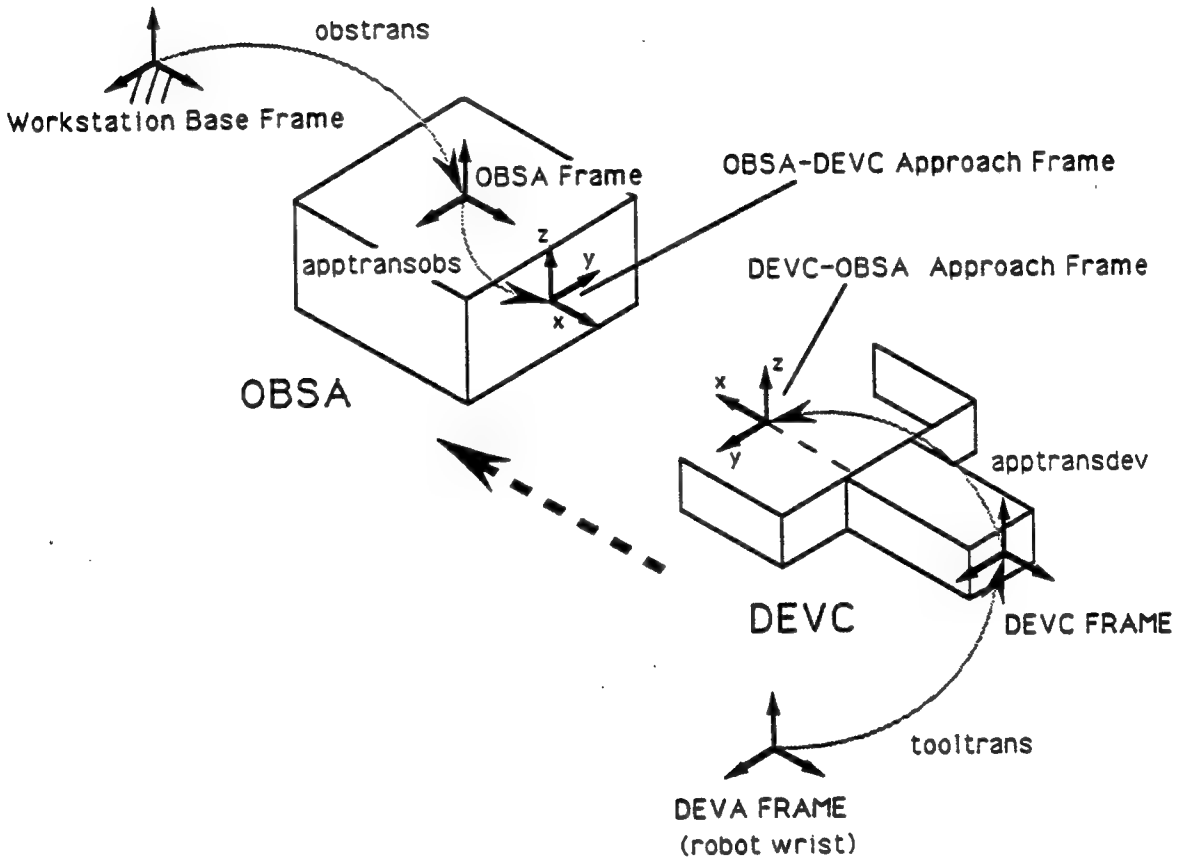


Figure B.9. DEVC Approaching OBSA

frames; the arrowheads at the end of these arcs define the direction of the transformations. Device frames are connected to the gripping device DEVC and the wrist of the robot DEVA (the robot is not shown in Figure B.9). The transformation represented by the variable-structure "obstrans" is specified between the Workstation Base Frame and the OBSA frame. The transformations "apptransobs" and "apptransdev" define the relationships between approach frames and the OBSA and DEVC frames respectively. The robot is positioned and orientated such that the x-axes of these approach frames are aligned as illustrated in the figure. The gripping device is translated along the x-axis of these aligned frames when making its grasping motion. The distance value "appdist" represents the initial displacement of the origins of the two approach frames along their x-axes. The linear grasping motion ends when the origins of the approach frames are made to coincide.

Functions such as intro_app_info()() demonstrate the flexibility gained by reducing task modules into functions and by the usage of FINITE-STATEs. The grasping procedures previously described would not change if they involved different objects and devices. The variations in object and device geometry, however, must be accounted for when determining the robot joint coordinates which produce the desired motion. FINITE-STATEs enable the separation of grasping procedures from the device- and object-specific information contained in the numerical transformations. The device- and

object-specific information is retrieved by the Planning Level software from data files (such as with "apptransobs" and "apptransdev") or from the operator (such as with "appdist").

The SS function intro_tooltrans()() (E0009) produces a structure "tooltrans" defining the transformation between the DEVC frame and the DEVA frame at the robot wrist (see Figure B.9). A series of transformations connecting the robot wrist frame to the Workstation Base Frame is now established. The function move_tool_toward()() (E0010) determines the robot joint coordinates for the grasping motion and places this data into the variable-structures "traj1" and "linetraj2". The data structure "resp12", contained in the output list of move_tool_toward()(), indicates whether or not the robot is capable of achieving the trajectories defined by "traj1" and "linetraj2". This data item is used by upcoming **pick_place_obs_to_dev()()** functions.

The function intro_sgdoub()() (E0011) and the seven SS functions following it compute the robot trajectories for the motion of DEVC as it lifts OBSA away from DEVD upon which it is currently residing. Figure B.10 depicts the transformations and frames related to the lifting motion. The intro_sgdoub()() function (E0011) outputs the data item "depdist" which contains the distance the robot should raise OBSA above DEVD. The functions intro_dvftrans()() (E0012) and intro_ddvob_info()() (E0013) yield the transformation for the DEVD frame ("devtrans") and the transformations for

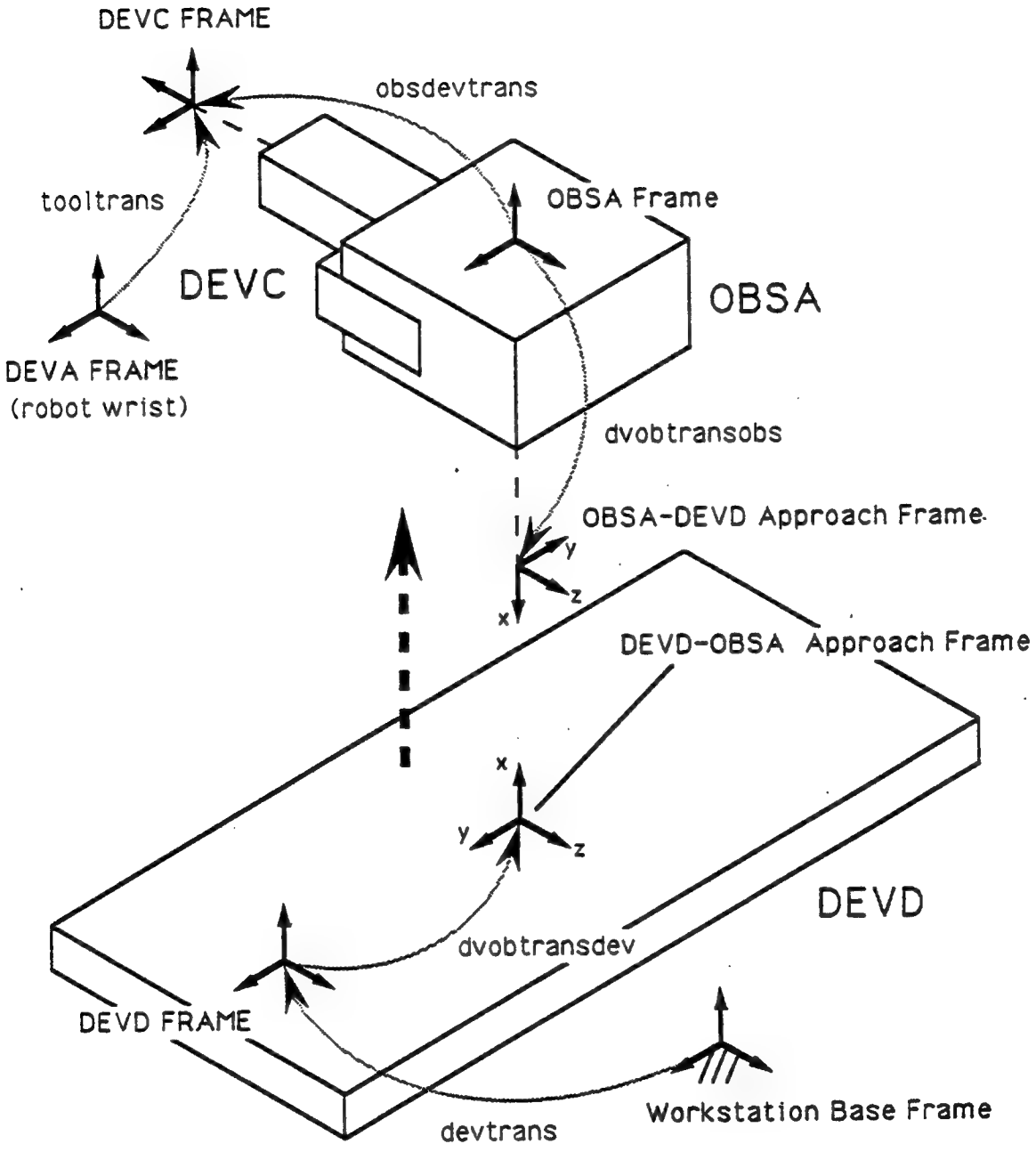


Figure B.10. DEVC Removing OBSA from DEVD

the approach frames between DEVD and OBSA ("dvobtransobs" and "dvobtransdev") respectively. The det_obsdevtrans()() function (E0016) calculates the transformation values represented in "obsdevtrans" which determine how OBSA is connected to DEVC. With all the transformations attained by these functions, the function move_obs_away()() (E0018) can determine the robot coordinates associated with the lifting trajectory. Trajectory values are placed into "linetraj4" while another output data structure "resp4" consists of a single value indicating the robot's ability (or inability) to perform the lifting trajectory.

Functions E0019 through E0029 are similar to those functions previously described except they determine robot motion trajectories for the remaining portion of the pick-and-place task. The function and_four_resp()() (E0031) examines the contents of variable-structures such as "resp12" and "resp4" which indicate whether the robot trajectories are achievable. The function produces a single data item "resp001" whose value indicates the success (or failure) of the entire pick-and-place task. This data structure is passed to the RC by the two communication functions at E0033 xmit_mssge_ss_rc()() and recv_mssge_ss_rc()(). The conditional functions in both the SS and RC listings: if001()() (E0035), else001()() (E0071), and endif()() (E0075), utilize "resp001" to determine if the pick-and-place task should be performed. The conditional functions divide two groups of functions, the first of which is executed if the robot is

capable of moving the gripping device along the computed trajectories. If it is determined that the robot is incapable of performing the desired motions, the module is terminated without directing any physical workstation activity.

Functions E0036 through E0070 implement the transfer of the robot trajectory data from the SS to the RC and command the robot to carry out its desired motion trajectories with action functions such as move_robot_jtintrp()() (E0041) and move_robot_strline()() (E0044). The SS action function activate_tool()() (E0047) activates the gripping mechanism of DEVC when this device is positioned adjacent to object OBSA. The synchronization function break()() is placed in the SS and RC listing before and after activate_tool()() to ensure that the robot complete its grasping motion prior to DEVC's gripping of OBSA and that DEVC has taken hold of OBSA before the robot lifts them both away from DEVD. The operation of synchronization functions is described with the following example. When the RC executes a break()() function, it outputs a signal to the SS computer and does not proceed with the next function in its listing until the RC receives a response signal from the SS. The corresponding break()() function in the SS (synchronization functions always come in pairs), when executed, realizes that the SS is being signaled by the RC, relays a signal to the RC, and continues with the execution of the next SS function. The last function in **pick_place_obs_to_dev()()** is exit_modpart()() (E0076) which converts the appropriate

process-level data structures into the task-level variable-structures contained in the module's output list.

Appendix C

User Interface and Graphics Simulation

The ODCAP software package presents information to and retrieves information from the workstation operator by means of user interface and graphics simulation software. The user interface software resides entirely in the planning computer termed the UIP (for User Interface and Planner) while graphics simulation programs are executed by the UIP and by the other planning computer known as the GS (for Graphics Simulator). The purpose and operation of these two important ODCAP features are reviewed in this appendix. Implementation details, such as the exact means by which graphics displays are mapped onto the monitor screen of the GS, are not considered; but a general understanding of how the user interface and graphics simulation software assist the operator during job planning is provided.

The user interface is menu-driven implying that the operator is continually provided with options by means of interactive screen menus. The operator selects those options which will cause the planning software to achieve his/her desired objectives. Figure C.1 shows a photograph of one such menu displayed on the monitor screen of the UIP. The contents and sequencing of the user interface menus correspond to the actions performed within the operator-assisted elements of the planning algorithm (see Chapter IV for descriptions of these elements). The operator is

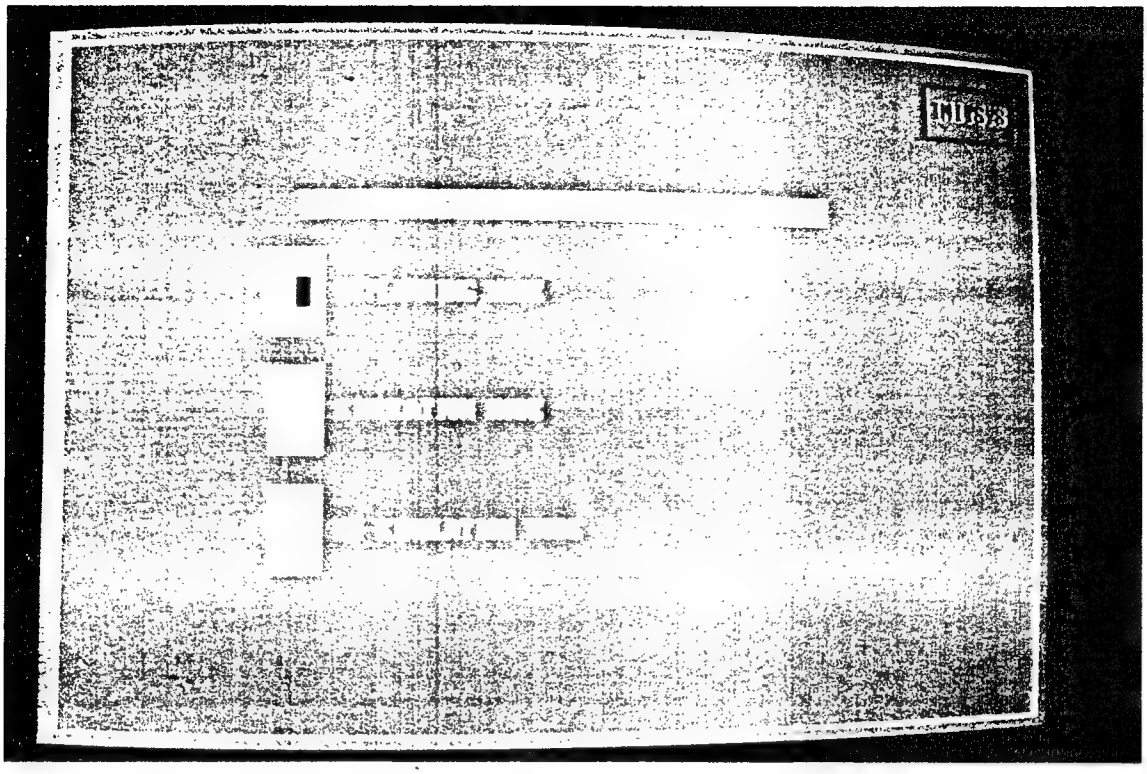


Figure C.1. User Interface Menu

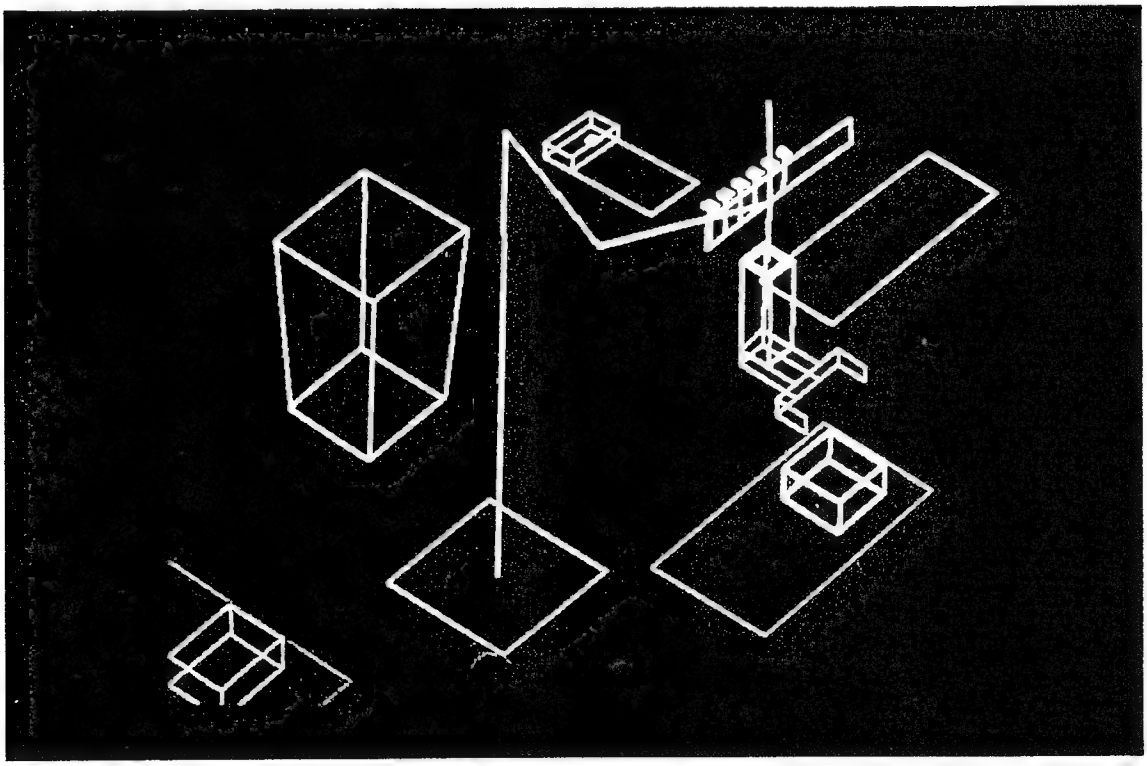


Figure C.2. Workstation Graphics Simulation

informed of the planning algorithm element to which each screen menu pertains. The user interface has two methods by which it obtains information from the operator, mouse selection and keyboard entry. When the operator must choose between two or more selections, he/she uses a mouse attached to the UIP to pick the desired selection. Keyboard entry is required when numerical or textual data must be entered. Keyboard-entered data is required for values such as the one representing the robot speed for the pick-and-place task presented in Appendix B. Task information such as robot speed cannot always be retrieved from data files and must be provided explicitly by the operator.

Every module has associated with it user interface software which enables the operator to specify the data needed for the insertion of a module into a job plan. This software is contained in the program files @.EXE located in the subdirectory \FCT (see Figure 4.1 on page 71 for a listing of all ODCAP files) where "@" denotes the shortened version of the module's name. The data retrieved from the operator (as well as the data retrieved autonomously from device and object files) is written to the introductory data section, the part of the job plan which contains all the data required for job plan execution. The user interface menus developed for each module behave identically to their planning algorithm counterparts. Planning algorithm menus require the text information placed in the data files \AAW\VIEW*.MEN (the "*" symbolizes all the names for these

menu text files) while module application menus require the information in \AAW\FCT\@.MEN. The information in the menu text files, indicated by the extension ".MEN", determines the visual appearance of the menus when they are displayed on the UIP monitor screen.

Graphics simulation software is developed for the workstation planning scheme to provide the operator with a "visual feel" of the workstation as it performs the activities corresponding to some job plan. Graphics simulation involves software in the UIP and the GS and requires some operator interaction. Graphics simulation software in the UIP and the GS is incorporated in the executable files \AAW\EXE\PLANNER.EXE and \AAW\EXE\SIMULATE.EXE respectively. When the planning algorithm is executed, communication software in these programs transfers "static" and "variable" graphics data from the UIP to the GS. Static graphics data represents the information required for the visual display of workstation devices and objects which does not change as the workstation performs tasks. This data is relayed to the GS only once at the start of a planning session. Variable graphics data includes visual display parameters which can be altered when tasks are performed. This data must be passed to the GS after a module is selected for inclusion into a job plan.

Workstation devices and objects are simulated using "stick-figure" representations as illustrated in the photograph provided in Figure C.2 which shows a typical graphics

display (projected onto the monitor screen of the GS). A simulation model of a robot with a gripper attached to its wrist is depicted in Figure C.3 for demonstration purposes. All workstation devices and objects are modeled using a three-dimensional coordinate system, representative of their physical dimensions. The coordinates of the line segments which make up the simulation models must be referenced, either directly or indirectly, to a single base coordinate system known as the Workstation Base Frame (as shown in Figure C.3). The means by which devices and objects are modeled is discussed in the "modeling" section of Chapter IV. Static graphics data for the robot model would include the three-dimensional coordinates defining the robot's link lengths and the numerical values indicating the model's various colors when it is displayed on the GS's monitor. The values representing the robot's joint coordinates are considered variable graphics data since they change when the robot undergoes simulated operation.

The monitor screen of the GS is a two-dimensional plane onto which the three-dimensional workstation simulation models are projected. ODCAP's graphics simulation software allows the operator to alter his/her viewing perspective of the workstation simulation so he/she can view the area where tasks are currently being performed. Figure C.4 depicts the manner in which the operator sees the three-dimensional workstation models as they are projected onto the GS monitor screen. This screen can be regarded as a viewing window

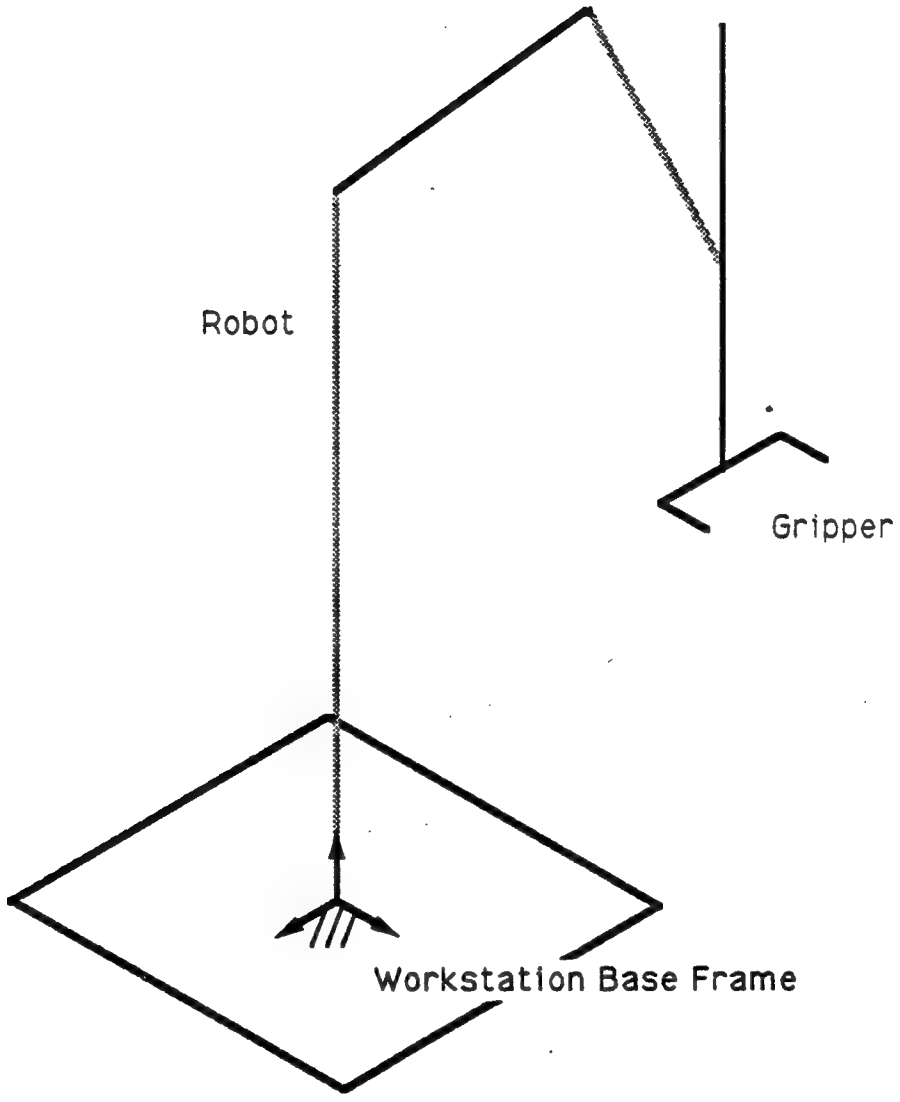


Figure C.3. Robot Simulation Model

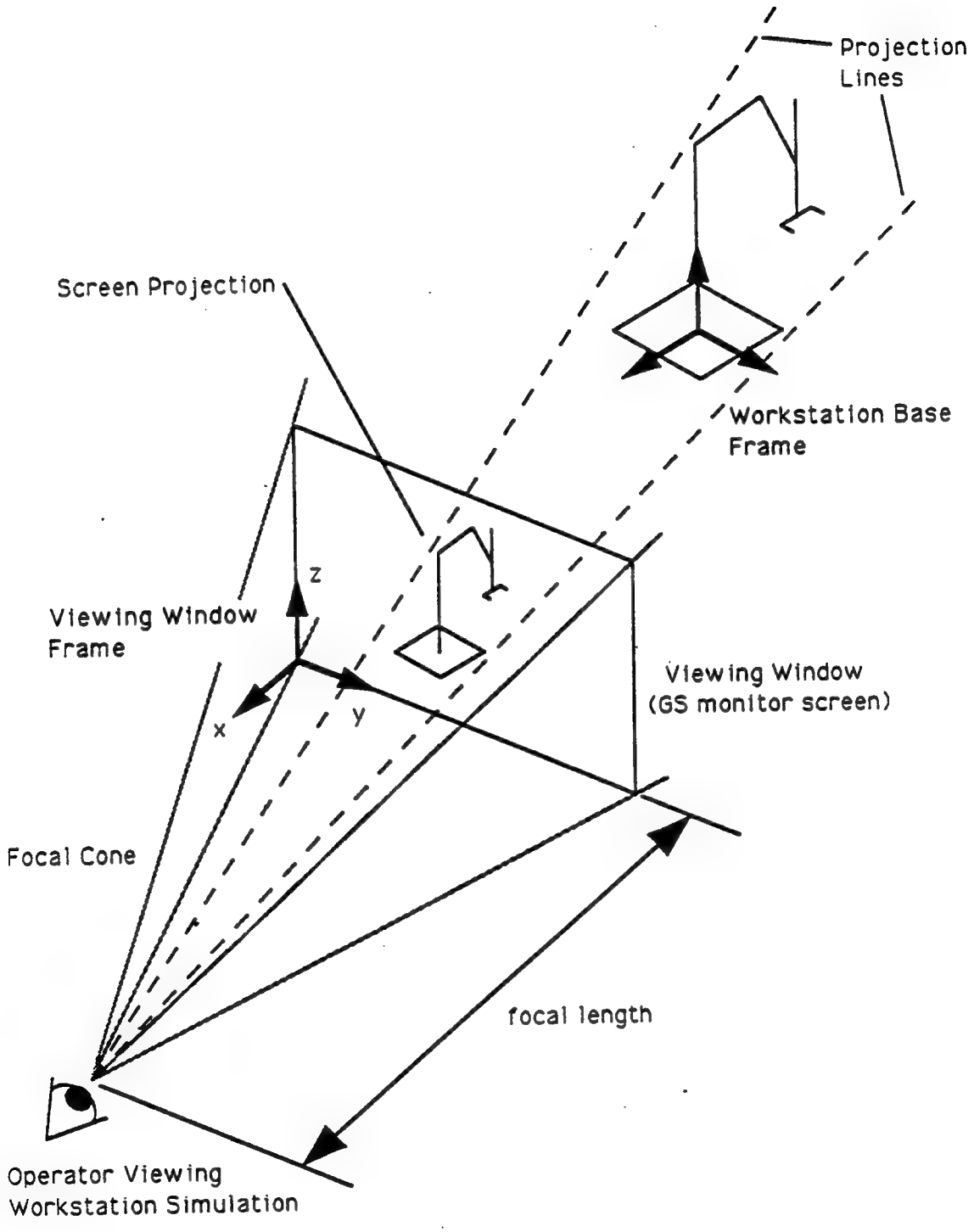


Figure C.4. Operator Viewing Graphics Simulation

through which the operator sees the workstation simulation. The operator's view is depicted as a single focal point positioned at the apex of a four-sided focal cone which is formed by the boundaries of the viewing window. A viewing window frame is attached to the lower corner of the window to define its position and orientation relative to the Workstation Base Frame. The x-axis of the viewing window frame coincides with the operator's line-of-sight; the viewing window resides on the yz-plane of the window frame. Mathematical transformations are used to determine the two-dimensional window frame coordinates of the three-dimensional simulation models as these models are projected, along the operator's line-of-sight, onto the viewing window.

Prior to the simulation of a workstation task, the operator can alter the position/orientation of the viewing window relative to the Workstation Base Frame. User interface menus permit the operator to translate the viewing window frame along any of its three axes (in either direction) and/or to rotate the frame about any of its axes (clockwise or counterclockwise). The operator can also specify a new focal length (the distance between the focal point and the viewing window) and/or change the horizontal/vertical dimensions of the viewing window. These changes would affect the shape of the focal cone and the scale of the workstation scene projected onto the viewing window. Those portions of the workstation model which fall

outside of the focal cone are not displayed on the GS monitor screen.

Simulation of a workstation task can involve up to nine stationary scenes reflecting the changes which the workstation devices and objects experience as the task is performed. The eight simulation scenes associated with the robotic pick-and-place task discussed in Appendix B are listed below, as an example.

Scene 1. Robot gripper is shown approaching the object it intends to acquire.

Scene 2. Robot gripper is shown adjacent to the object prior to grasping it.

Scene 3. Robot gripper is shown after it has grasped the object.

Scene 4. Robot gripper is shown after it has lifted the object away from the device that previously held the object.

Scene 5. Robot gripper is shown approaching the device onto which it intends to place the grasped object.

Scene 6. Robot gripper is shown adjacent to the object prior to releasing it.

Scene 7. Robot gripper is shown after it has released the object.

Scene 8. Robot gripper is shown after it has moved away from the object.

Each scene is displayed briefly (about 1-3 seconds) on the GS monitor screen enabling the operator to view the activities connected to job plan tasks. The designation of simulation scenes is performed by a programmer when he/she creates the task-level software associated with module implementation in the program files \AAW\FCT\@.EXE. Sensing

action modules, conditional module sets, and computational modules do not require the development of simulation software since the workstation remains unchanged after the tasks corresponding to these modules are performed. No information is communicated back to the UIP from the GS during graphics simulation. The graphics feature of the ODCAP software package does not affect job planning and is only intended as "visual feedback" for the workstation operator.

Appendix D

Rule-Base Facts and Rules

Examples of the facts and the rules which are used by the planning scheme's rule-base systems are provided in this appendix. The organization and purpose of the three rule-bases known as the ENSFRB, the EMRRB, and the IRB are presented in Figure 3.11 on page 60 and in the accompanying text of Chapter III. The syntactical form of every kind of fact is presented in this appendix for the following fact types: Explicit Generic "State" (EGS) facts, "history" facts, implicit facts, and Possible Module (PM) facts. Explicit Non-generic "State" (ENS) facts and the rules used by each of the three rule-bases do not possess strict syntactical forms; a single example for each of these items is provided to illustrate their composition.

EGS facts are utilized to store rule-base information related to a workstation's status. The six kinds of EGS facts are listed in Figure D.1 in predicate calculus form. The various properties associated with every fact are listed between the parentheses adjacent to the fact's label. Fact properties are defined by the terms contained within brackets. The EGS fact **device** (fact labels are denoted by boldfaced, lower-case characters) describes workstation devices and has five properties connected to it. The first two properties specify the device's name and identification number. The device's class and subclass (see the resource

Explicit Generic State Facts

```
device((device name), (device identification number),
       (device class), (device subclass), (movability))

object((object name), (object identification number),
       (pose known value), (object class), (object subclass),
       (object form number))

switch((device identification number), (switch name), (switch state))

dev_attch((identification number for movable device),
          (identification number for coupling device),
          (device coordinate frame for movable device),
          (device coordinate frame for coupling device))

obs_attch((object identification number),
          (device identification number))

obs_asmbly((assembly name), (assembly identification number),
           (number of objects in assembly),
           (list of identification numbers for assembly objects),
           (identification number for coupling device))
```

"History" Facts

```
outpvar((variable-structure name), (device name), (computer label))

num_nested_condls((number of nested conditionals))

inside_condl((nesting index number), (inner block number),
             (conditional module set name))
```

Implicit Facts

```
inptvar((variable-structure name), (device name), (computer label))

outpvar((variable-structure name), (device name), (computer label))

num_implmod((number of implicit modules applied to plan))

implmod((implicit module identification number),
        (implicit module name), (nth FINITE-STATE), (n),
        (top of plan indicator))

sem((SEM name), (nth FINITE-STATE), (n))
```

Possible Module Fact

```
posmod((explicit module name), (instance number),
       (nth FINITE-STATE), (n))
```

Figure D.1. Rule-Base Facts

classifications listed in Figure 4.2 on page 78) are represented in the third and fourth fact properties. The last property termed the "movability" can take on the values "movable" or "fixed", indicating whether the device is capable (or not) of being re-attached to some other device. An example of a **device** fact is provided as follows:

device(hand, 03, tool, gripping, fixed)

This fact describes a device (labeled "hand") which is attached to a robot wrist and is used as a gripping tool. The property "fixed" does not imply that the device "hand" is immovable but that it is permanently attached to some other device.

The properties of the EGS fact **object** are similar to those of **device** except for the two properties indicated as the "pose known value" and the "object form number". The property "pose known value" takes on the value "1" if the position and orientation of the object described by the fact are known and is assigned the value "0" if these object characteristics are unknown. This property can indicate to the rule-bases the need to apply a sensing task to determine an object's location. Objects can have their physical attributes altered by the tasks performed during the execution of a job plan. The **object** fact property "object form number" can be used to differentiate between the various "forms" the object can take on as it is acted on by workstation devices.

The EGS fact **switch** defines the status of device switches (see the "modeling" section in Chapter IV for a definition of device switches). The fact labeled **dev_attch** provides information regarding "movable" devices. The device referred to (in the property definitions) as a "coupling device" is the device to which a "movable" device is attached. The fact **obs_attch** specifies the device to which each object is currently attached. Every "movable" device or object requires the presence of one **dev_attch** fact or one **obs_attch** fact, respectively, in the Explicit Data Base (EDB). The EGS fact **obs_asmbly** incorporates information on workstation assemblies.

Three kinds of "history" facts are included in Figure D.1; these facts define important qualities pertaining to the job plan which are needed by the planning algorithm. The fact **outpvar** specifies the output variable-structures produced when a module is added to a job plan. If, for example, the module defined as follows:

```
pick_and_place_obs_to_dev
(BLOCK, ROBOT, TOOLGRP, GRIPPER,
 CONVEYOR, TABLE, ndevvar:ROBOT:SS,
 devvari:ROBOT:SS, nobsatc:CONVEYOR:SS,
 obsatci:CONVEYOR:SS, nobsatc:TABLE:SS,
 obsatci:TABLE:SS)
(ndevvar:ROBOT:SS, devvari:ROBOT:SS,
 nobsatc:CONVEYOR:SS, obsatci:CONVEYOR:SS,
 nobsatc:TABLE:SS, obsatci:TABLE:SS)
```

is inserted into a Master Plan, the following **outpvar** facts would be asserted to the appropriate data bases:

```
outpvar(ndevvar, robot, ss),
outpvar(devvari, robot, ss),
outpvar(nobsatc, conveyor, ss),
outpvar(obsatci, conveyor, ss),
outpvar(nobsatc, table, ss), and
outpvar(obsatci, table, ss).
```

The other two "history" facts labeled **num_nested_condls** and **inside_condl** describe the "nesting" of conditional module sets in the job plan. The "nesting" of conditional modules refers to the placement of conditional module sets between the modules of other conditional module sets. If the planning algorithm, as an example, is currently inserting modules within the conditional module set labeled WHLOBDEV whose own modules are contained within a module set named RPTCOUNT, the following "history" facts would be designated to define job plan conditions:

```
num_nested_condls(2),
inside_condl(1, 1, rptcount), and
inside_condl(2, 1, whlobdev).
```

The **inside_condl** fact property referred to as the "inner block number" indicates which section of the conditional the "nested" modules are situated. The value for this property is always "1" except for IF-ELSE-ENDIF, IF-ELSEIF-...-ENDIF, and IF-ELSEIF-...-ELSE-ENDIF conditionals (see Appendix A for information on conditional module sets).

Five kinds of facts comprise the implicit facts and as listed in Figure D.1. The facts **inptvar** and **outpvar** define the respective input and output variable-structures produced

when the planning algorithm attempts to insert implicit modules into a job plan before an operator-selected module, known as the Selected Explicit Module (SEM), to make the plan workable. The insertion operation is performed autonomously by the Implicit Rule-Base (IRB) system and is explained in detail in Chapter IV. The facts **implmod** and **sem** define the characteristics of the inserted implicit modules and the SEM respectively, including the values of the modules' FINITE-STATEs. The last property of the fact **implmod** fact termed the "top of plan indicator" is assigned the value "1" if the implicit module is to be inserted at the top of a job plan and is designated as "0" if the implicit module is intended for Master Plan insertion at a location just above the SEM.

Possible Module (PM) facts are used by the Explicit and Implicit Rule-Base systems to define all the possible modules (explicit) which can be applied to a job plan. Only one kind of PM fact (see Figure D.1) is required to define a possible module. The manner by which the PM fact **posmod** defines possible modules is explained with the use of an example. If a module named PPOBSDEV possessing the FINITE-STATE values (listed in the order for which they appear in the module's input list): BLOCK, ROBOT, TOOLGRP, GRIPPER, CONVEYOR, and TABLE, is determined to be a possible module,

the following six PM facts may be asserted to the EDB and the Implicit Data Base (IDB):

```
posmod(ppobsdev, 05, block, 01),
posmod(ppobsdev, 05, robot, 02),
posmod(ppobsdev, 05, toolgrp, 03),
posmod(ppobsdev, 05, gripper, 04),
posmod(ppobsdev, 05, conveyor, 05), and
posmod(ppobsdev, 05, table, 06).
```

The second property for each fact, the value "05", indicates that these facts refer to the fifth instance of a possible module involving PPOBSDEV. Other PM facts associated with the module PPOBSDEV with different "instance numbers" define possible modules which have different FINITE-STATE values.

Explicit Non-generic "State" (ENS) facts and the rules comprising the ENSFRB, the EMRRB, and the IRB are not limited to established forms as are the facts shown in Figure D.1. Rules assert new facts based on "existent" data base facts and are created using the Prolog programming language. Clocksin and Mellish [73] provide the definitive reference for the operation and usage of the Prolog programming language. This appendix provides one example of an ENSFRB rule, an EMRRB rule, and an IRB rule, shown in Prolog code exactly as they would appear in the programs which implement the rule-bases. The example ENSFRB rule incorporates an example of an ENS fact.

Figure D.2 lists the three Prolog rule examples; the ENSFRB rule, the EMMRB rule, and the IRB rule are identified by the respective three-digit identification values: "f002", "r004", and "i001" (all rule-base rules are marked in this

ENSFRB Rule Example

```
f002 :-
    repeat, !,
    call(object(Obs1,Obs1id,1,_,_,_)),
    call(device(Dev,Devid,_,_,_)),
    call(device(table,Tabid,_,_,_)),
    call(obs_attch(Obs1id,Devid)),
    Devid \= Tabid,
    call(object(Obs2,Obs2id,_,_,_,_)),
    not(call(obs_attch(Obs2id,Tabid))),
    not(call(do_move_operation(Dev,table,Obs1))),
    assertz(do_move_operation(Dev,table,Obs1)),
    fail.
```

EMRRB Rule Example

```
r004(File) :-
    repeat, !,
    call(posmod(ppobsdev,Inst,ObsA,1)),
    call(posmod(ppobsdev,Inst,robot,2)),
    call(posmod(ppobsdev,Inst,DevB,3)),
    call(posmod(ppobsdev,Inst,DevC,4)),
    call(posmod(ppobsdev,Inst,DevD,5)),
    call(posmod(ppobsdev,Inst,DevE,6)),
    DevB \= DevC,
    call(do_move_operation(DevD,DevE,ObsA)),
    not(call(recom004(ObsA,DevB,DevC,DevD,DevE))),
    assertz(recom004(ObsA,DevB,DevC,DevD,DevE)),
    write(File,$* R004 *$), nl(File),
    string_term(RecPar1,ObsA), write(File,RecPar1), nl(File),
    string_term(RecPar2,DevB), write(File,RecPar2), nl(File),
    string_term(RecPar3,DevC), write(File,RecPar3), nl(File),
    string_term(RecPar4,DevD), write(File,RecPar4), nl(File),
    string_term(RecPar5,DevE), write(File,RecPar5), nl(File),
    fail.
```

IRB Rule Example

```
i001(File) :-
    repeat, !,
    call(posmod(indevvar,_,Dev,1)),
    call(inptvar(ndevvar,Dev,ss)),
    call(inptvar(devvari,Dev,ss)),
    not(call(implmod(_,indevvar,Dev,01,1))),
    write(File,$indevvar$), nl(File),
    write(File,$01$), nl(File),
    string_term(FS1,Dev), write(File,FS1), nl(File),
    write(File,$1$), nl(File).
```

Figure D.2. Prolog Rule Examples

manner). The ENSFRB example attempts to assert the ENS fact **do_move_operation** based on information derived from EGS facts **object**, **device**, and **obs_attch**. The term "call", used frequently in all three rule examples, represents a Prolog operator which attempts to match known facts to a given set of conditions. For example, the statement:

```
call(object(Obs1,Obs1id,1,_,_,_)),
```

which constitutes the second line of the ENSFRB rule will cause the rule-base to search for **object** facts with properties "Obs1" and "Obs1id" (property names beginning with a capitalized letter indicate that the property value is variable) where the facts indicate an object with known position and orientation. The "_" character is used to signify that any value can be assigned to the given fact property. The ENSFRB rule example asserts a **do_move_operation** fact if an object "Obs1" can be discovered with known position/orientation which is attached to a device "Dev". The device to which "Obs1" is currently attached cannot be the device "table". The asserted fact **do_move_operation** specifies to the EDB that an operation can be performed on "Obs1" which results in the relocation of "Obs1" from "Dev" to "table". The "repeat" term indicates that the rule will search the entire EDB to find facts which meet its conditions and which permit it to specify new **do_move_operation** facts.

The EMRRB rule example identified as "r004" makes a module recommendation for the explicit module PPOBSDEV if

the rule can obtain facts to support its numerous conditions. The first six "call" statements in the rule determine the possible values of FINITE-STATEs which are currently applicable to PPOBSDEV. Another "call" statement searches for the ENS fact **do_move_operation** to obtain the FINITE-STATE names of the devices and object to which the recommended PPOBSDEV module should apply. The remaining rule statements assert the recommendation associated with PPOBSDEV to the explicit module recommendations.

The IRB rule (marked "i001") can insert the module INDEVVAR into a Master Plan if the input variable-structures: ndevvar:DEV:SS and devvar:DEV:SS (DEV indicates a FINITE-STATE representing some device), are located in the input data list of the Selected Explicit Module (SEM). The property value "1" in the **posmod** fact contained in the rule's first "call" statement indicates that the module INDEVVAR is to be inserted at the top of the job plan. This rule ensures that any module requiring information incorporated in the two variable-structures: ndevvar:DEV:SS and devvar:DEV:SS, can be applied to the job plan if an INDEVVAR module can be determined which outputs these data structures.

Appendix E

Planning Algorithm Pseudocode

The algorithm which implements the semi-autonomous planning scheme for workstations (see Chapter IV) has its operations divided into eleven divisions of program procedures, known as "code blocks". This appendix describes the algorithmic operations which occur in each code block with the use of "pseudocode". Pseudocode utilizes English-like sentences to denote program actions and to convey the meaning of these actions.

The planning algorithm is implemented by the executable program \AAW\EXE\PLANNER.EXE (see Figure 4.1 on page 71 for listings of all computer files). This program is developed using Microsoft C and is executed in the planning computer, known as the UIP, simultaneous with the execution of the program \AAW\EXE\SIMULATE.EXE in the computer which implements graphics simulation, termed the GS. The pseudocode, presented in this appendix, reveals the interactions which take place between the UIP program and the GS program (also developed using Microsoft C). The purpose of the GS program is to receive workstation simulation data from the UIP and to generate and to display the appropriate graphics on the GS monitor screen when instructed by the UIP.

The planning algorithm flowchart is reproduced from Chapter IV in Figure E.1 while the pseudocode is listed for each of the eleven code blocks in Figures E.2 through E.9.

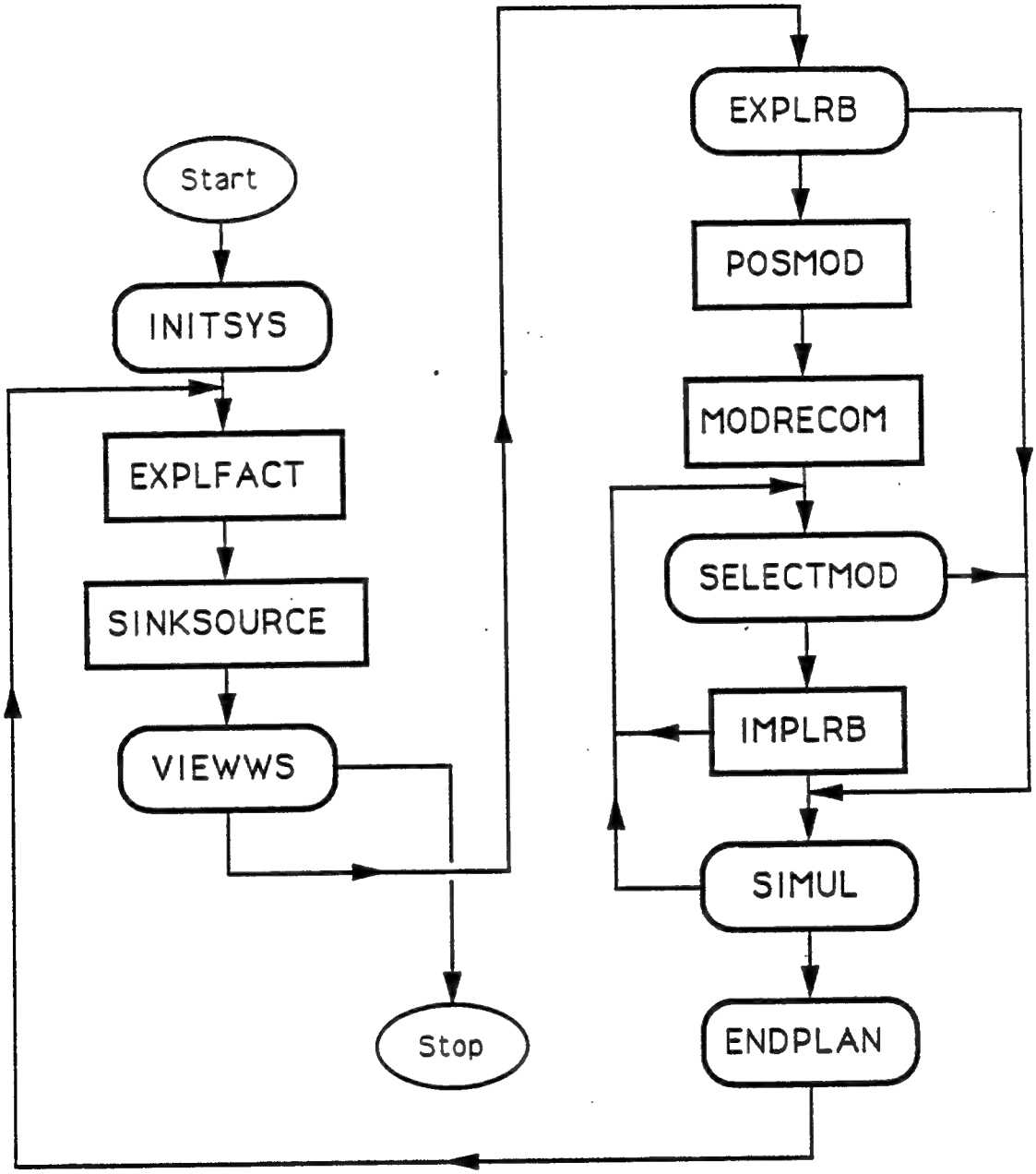


Figure E.1. Planning Algorithm Flowchart

```
INITSYS:

Set first_time_thru_efact = 1.

Set first_time_thru_simul = 1.

Set first_time_thru_psmod = 1.

Set first_time_thru_IRB = 1.

Set inside_inner_block = 0.

Set insert_module = 0.

Set user_end_plan = 0.

Query operator for values of plan_mode and
  plan identification number.

Initialize global device parameters.

IF plan_mode = 0 THEN

   Set is_module_known = 0.

   Initialize Master Plan and
     introductory data section.

ELSE

   Set is_module_known = 1.

   Execute implicit modules at top of plan.

   IF plan_mode = 3 THEN

      Initialize introductory data section.

   ENDIF

ENDIF

Pass static simulation data to GS.

GOTO EXPLFACT.
```

Figure E.2. Pseudocode for INITSYS

```
EXPLFACT:

IF first_time_thru_efact = 1 THEN

   Determine "new" EGS facts from
     global device parameters.

   Initialize EDB with "new" EGS facts.

   IF plan_mode != 0 THEN

      Determine "history" facts.

   ENDIF

   Set first_time_thru_efact = 0.

ELSE

   Determine "new" and "old" EGS facts from
     global device and object parameters.

   Assert "new" EGS facts to and retract
     "old" EGS facts from EDB.

ENDIF

GOTO SINKSOURCE.
```

Figure E.3. Pseudocode for EXPLFACT

```
SINKSOURCE:

Sink departing objects and assemblies and
  source incoming objects.

Determine "new" and "old" EGS facts from
  global object parameters.

Assert "new" EGS facts to and retract
  "old" EGS facts from EDB.

GOTO VIEWWS.
```

```
VIEWWS:

IF first_time_thru_simul = 1 THEN

   Determine first initial variable simulation data.

   Pass variable simulation data to GS.

ENDIF

Permit operator to alter viewing perspective
  of workstation graphics simulation.

IF first_time_thru_simul = 0 THEN

   Pass variable simulation data to GS.

   Instruct GS to begin simulation of task.

ENDIF

Set first_time_thru_simul = 0.

IF user_end_plan != 0 THEN

   Quit planning program.

ENDIF

GOTO EXPLRB.
```

Figure E.4. Pseudocode for SINKSOURCE and VIEWWS

```
EXPLRB:

Assert "history" facts to EDB.

Determine ENS facts by activating ENSFRB.

IF plan_mode = 1 AND is_module_known = 1 THEN

   Query operator for value of insert_module.

   IF insert_module = 1 THEN

      Set is_module_known = 0.

      GOTO POSMOD.

   ENDIF

ENDIF

IF is_module_known = 1 THEN

   Assign values to SEM and its corresponding
     implicit modules from Master Plan.

   GOTO SIMUL.

ENDIF

GOTO POSMOD.
```

```
POSMOD:

IF first_time_thru_psmod = 1 THEN

   Determine PM facts based on device-related
     PM requirements.

   Set first_time_thru_psmod = 0.

ENDIF

Determine PM facts based on device- and
  object-related PM requirements.

GOTO MODRECOM.
```

Figure E.5. Pseudocode for EXPLRB and POSMOD

```
MODRECOM:

Assert PM facts to EDB.

Determine explicit module recommendations
  by activating EMRRB.

IF inside_inner_block = 1 THEN

   Add next conditional module located below pointer
     in Master Plan to explicit module recommendations.

ENDIF

GOTO SELECTMOD.
```

```
SELECTMOD:

Query operator regarding SEM by providing
  him/her explicit module recommendations.

Determine initial implicit facts.

Assign appropriate value to input_output_data_matched
  if SEM input data requirements can or cannot be
  satisified without implicit module.

IF input_output_data_matched = 1 THEN

   Assign values to SEM from initial implicit facts.

   GOTO SIMUL.

ENDIF

GOTO IMPLRB.
```

Figure E.6. Pseudocode for MODRECOM and SELECTMOD

```
IMPLRB:

IF first_time_thru_IRB = 1 THEN

   Initialize IDB with PM facts and
     initial implicit facts.

   Set first_time_thru_IRB = 0.

ELSE

   Assert PM facts and initial implicit facts to IDB.

ENDIF

REPEAT

   Determine "new" implicit facts by activating IRB.

   Assign appropriate value to IRB_exhausted if
     IRB can or cannot determine implicit module
     for satisfying SEM input data requirements.

   IF IRB_exhausted = 1 THEN

      Break out of REPEAT-UNTIL loop.

   ENDIF

   Assign appropriate value to input_output_data_matched
     if SEM input data requirements can or cannot be
     satisified with addition of new implicit module.

   Determine "old" implict facts.

   Assert "new" implicit facts to and retract
     "old" implicit facts from IDB.

UNTIL input_output_data_matched = 1

IF IRB_exhausted = 1 THEN

   Remove SEM from explicit module recommendations.

   GOTO SELECTMOD.

ENDIF

Assign values to SEM and its corresponding
  implicit modules from implicit facts.

GOTO SIMUL.
```

Figure E.7. Pseudocode for IMPLRB

```
SIMUL:

Execute implicit modules.

Execute SEM and assign appropriate value to
  retval based on success or failure of SEM execution.

IF is_module_known = 1 THEN

   Adjust location of pointer in Master Plan.

   IF plan_mode = 3 THEN

      Add new data to introductory data section.

   ENDIF

ELSE

   Query operator for value of userval.

   IF userval = 1 AND retval = 1 THEN

      Add new data to introductory data section.

      Insert SEM and its corresonding implicit
        modules into Master Plan.

      Adjust location of pointer in Master Plan.

   ELSE

      GOTO SELECTMOD.

   ENDIF

ENDIF

Assign appropriate value of is_module_known by
  examining pointer location in Master Plan.

Assign appropriate value of inside_inner_block if
  pointer is currently located within or not
  within any conditional module set.

Determine "history" facts.

GOTO ENDPLAN.
```

Figure E.8. Pseudocode for SIMUL

```
ENDPLAN:
IF plan_mode = 2 OR plan_mode = 3 THEN
   IF is_module_known = 0 THEN
      Set user_end_plan = 1.
   ENDIF
ELSE
   Query operator for value of user_end_plan.
   IF user_end_plan = 1 THEN
      Remove pointer from Master Plan.
   ELSEIF user_end_plan = 2 THEN
      Remove pointer from Master Plan.
      Generate subplans from Master Plan.
   ENDIF
ENDIF
GOTO EXPLFACT.
```

Figure E.9. Pseudocode for ENDPLAN

Pseudocode actions are structured in "verb-subject" format as sentences such as "Assert PM facts to EDB". The execution sequence of pseudocode actions is dependent on the integer values assigned to data terms known as "algorithmic variables". These variables are used by conditional structures similar to the IF-ELSE-ENDIF and REPEAT-UNTIL conditional module sets discussed in Appendix A. An example of the manner by which an algorithmic variable can alter the execution sequence is provided using the following pseudocode actions.

```
IF user_end_plan != 0 THEN
  Quit planning program.
ENDIF
```

The pseudocode action "Quit planning program" is only performed if the algorithmic variable "user_end_plan" is assigned to some value other than zero (the symbol "!=" should be read as "not equal to"). Conditional terms such as "IF" and "ENDIF" are capitalized to distinguish them from other pseudocode actions. Branching between code blocks is implemented using the pseudocode action "GOTO **BLOCKNAME**" where **BLOCKNAME** refers to one of the code block labels. Examination of the pseudocode will reveal that the code block branching brought about by GOTO actions corresponds to that of the planning algorithm flowchart in Figure E.1.

The purpose and operation of the algorithm code blocks are discussed in Chapter IV and are not restated in this appendix. Comprehension of the pseudocode requires that the

reader understands all the procedures associated with the ODCAP planning algorithm. Definitions for every algorithmic variable used in the pseudocode are provided since these data terms influence the order in which pseudocode actions are performed. The first four pseudocode actions in **INITSYS** concern the assignment of values to four different algorithmic variables whose names all begin with the characters "first_time_thru_". These variables when set to equal "1" indicate that certain initialization actions must be performed; they are assigned the value "0" after these actions are conducted. For example, if "first_time_thru_efact" is set to "1" in **EXPLFACT**, the initialization of the EDB is performed. This action need only be carried out once per planning session so the variable is set to "0" following EDB initialization.

The algorithmic variable "inside_inner_block" is set to "1" if the pointer, which tracks the execution of the modules within a Master Plan, is currently located between the modules of a conditional set (see Appendix A for information on conditional modules) and is set to "0" if the pointer is not situated between conditional modules. The "plan_mode" variable denotes the planning mode and is set by the operator in **INITSYS**. This variable can take on the following values, correspondent to the four planning modes: "0" for regular planning, "1" for replanning, "2" for simulation, and "3" for re-simulation. The "is_module_known" variable

is set to "1" or "0" if the next module scheduled for execution is known or unknown respectively.

The operator specifies his/her intention to terminate the planning session through the use of the "user_end_plan" algorithmic variable. This variable is set to "0" until the operator instructs the algorithm to cease planning; it then can acquire the value "1" or "2" dependent on the operator's desire to omit or to carry out, respectively, the creation of the process-level subplans. The variable "insert_module" is utilized in the code block **EXPLRB** to indicate if the operator wishes to insert (when set to "1") or not to insert (when set to "0") a new module into the plan. The code blocks **SELECTMOD** and **IMPLRB** make use of the variable "input_output_data_matched" to determine if the input data requirements of the newly-inserted explicit module are satisfied (when set to "1") or unfulfilled (when set to "0"). The variable "IRB_exhausted" is set to "1" if the IRB fails to produce an implicit module which assists in satisfying these input data requirements. The algorithmic variables "userval" and "retval" are used in **SIMUL** to define the operator's desire to insert a new module into a plan (when "userval" is set to "1") and the success or failure (when "retval" is assigned the respective value of "1" or "0") of the newly-inserted module's simulated execution, respectively.

Appendix F

Object Sinking and Sourcing

The terms "sinking" and "sourcing" are used to designate the removal of objects from and the entry of objects into the workstation environment respectively. Manufacturing workstations are designed to perform operations on objects so as to transform them into useful products. Since workstation equipment is assumed stationary, a means for moving objects into and out of the workstation must be devised. Any planning scheme must take into account the sourcing and sinking of objects when deciding the activities necessary to the performance of a workstation job.

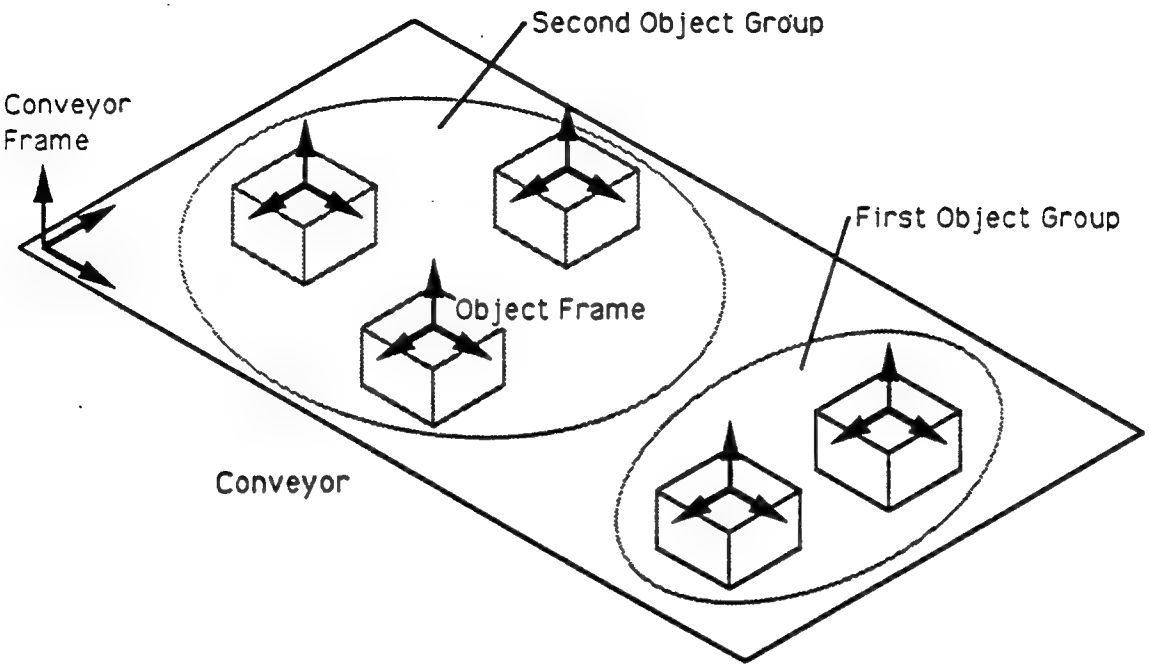
The various methods by which workstation devices acquire new objects and eject old objects and assemblies must be considered. The ODCAP software implements two such methods for sourcing objects and one for sinking objects and assemblies. These methods are derived from observations of the operation of actual sourcing and sinking devices. New sourcing and sinking methods can be developed for ODCAP to implement techniques different from the three presented in this appendix.

The software which automatically performs sinking and sourcing operations during job planning is contained in the program code block known as **SINKSOURCE**. An object or assembly is "sinked" after it becomes attached to a device possessing the classification subsubclass known as sink (see

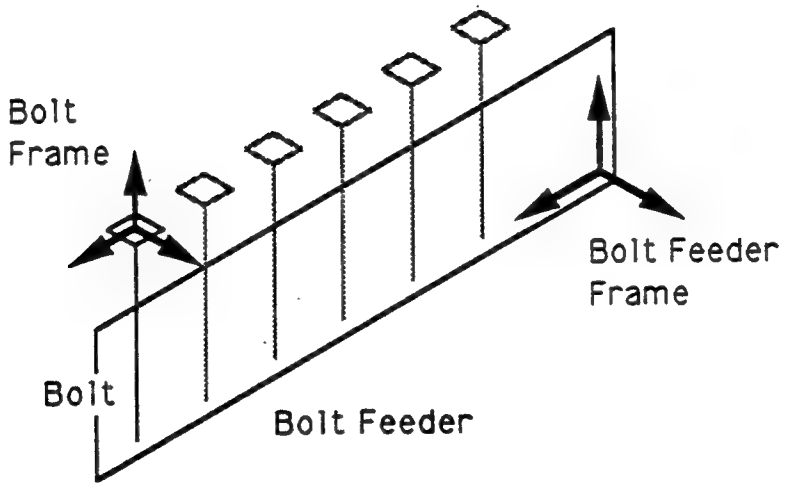
Figure 4.2 on page 78 for the resource classifications of all workstation devices). Sinking involves the elimination of all the global parameter data associated with the object or assembly undergoing removal. The global parameters are a data set which define workstation properties such as object positions and orientations (henceforth referred to as poses). Global object parameter data is reordered following the sinking of an object or assembly to prevent ODCAP from confusing those objects which remain in the workstation environment with those objects which have been removed. An object or assembly should be relocated to a device with subsubclass sink only after all manufacturing operations have been performed on it.

The two sourcing methods involve devices which bring objects into the workstation with known and unknown poses respectively. The first method constrains objects in such a manner that their poses remain fixed while the other method sources objects randomly without geometric constraints. Example devices and procedures are provided for both methods to illustrate their operation.

A linear conveyor is supplied as an example of a device which sources objects with unknown poses. ODCAP requires that sourcing devices which implement this method are classified as class mover and subsubclass sink. Figure F.1(a) depicts a conveyor with block-like objects randomly situated on it. Each object possesses a coordinate frame which is



(a) Sourcing Objects with Unknown Poses



(b) Sourcing Objects with Known Poses

Figure F.1. Object Sourcing Examples

used to define the geometric relationship between the incoming object and the conveyor coordinate frame. The transformation values which determine these relationships must be specified by the operator before planning can begin (see Appendix G for ODCAP initialization procedures).

Objects attached to a source device are partitioned into groups; any number of groups or objects within a group may be specified by the operator. When ODCAP planning is initiated, global parameters are defined automatically for all the objects contained in the first group. These objects are described as being "sourced" into the workstation. The "unsourced" objects attached to a source device (such as the conveyor) are depicted by the graphics simulation but are not yet considered in the workstation environment. A single global parameter is used to indicate that the pose of each object in the first group is unknown. The ODCAP planning algorithm may respond to this parameter setting by recommending that a sensing task be performed to determine the poses of the sourced objects. The objects in the second group will not be sourced until all the objects in the first group are removed from the source device. This fact applies to the second group as well as any subsequent object group attached to the source device.

Sourcing objects with known poses is demonstrated using a bolt feeder device illustrated in Figure F.1(b). This sourcing method is only conducted for source devices of

class fixture, subclass queue, and subsubclass source. Incoming objects are organized into groups just as they are with the method involving objects with unknown poses. Every group in the bolt feeder consists of a single object fastener bolt. Global parameter data is acquired for the first bolt when ODCAP planning is begun by the operator. The design of the bolt feeder constrains the sourced bolt such that its position and orientation can be considered as known. Global parameter data for the second bolt is defined after the first bolt has been removed from the bolt feeder.

Another significant difference between the two sourcing methods relates to the manner in which source devices, such as the bolt feeder, alter the poses of the objects attached to them after all the objects in the "sourced" group have been removed. It is assumed that source devices such as the bolt feeder possess some mechanism (such as a spring or gravity) which advances the objects attached to them to different positions/orientations when the objects in the "sourced" group are removed by other workstation devices. For the bolt feeder example, when the first bolt is removed, the second bolt relocates to the pose originally held by the first bolt, the third bolt relocates to the old pose of the second bolt, and so forth. The automatic advancing of objects for this sourcing method means that the pose of every sourced object remains constant in value. ODCAP software has been created in the code block **SINKSOURCE** which implements these sourcing procedures whenever devices

are classified with the appropriate class, subclass, and subsubclass.

Appendix G

Guide to Using ODCAP

This appendix addresses the various actions a workstation operator must perform to effect job planning using the ODCAP software package. The manner in which the information files associated with ODCAP are created is not discussed. A programmer (or programmers) who is familiar with ODCAP must construct these files which include information on ODCAP entities such as modules, rules, devices, and objects. All ODCAP files are assumed to be situated in subdirectories on the planning computers known as the UIP and the GS, as shown in Figure G.1. This figure is reproduced from Chapter IV which provides descriptions of nearly every ODCAP file.

Before job planning can begin, the operator must specify the workstation configuration, the initial device data, and the incoming object data. The operator executes the program \AAW\EXE\SETUP.EXE to designate the workstation configuration. This program has interactive menus (see Appendix C for information on ODCAP's user interface and graphics simulation capabilities) which query the operator for the names of the devices and objects that job planning is to involve. Once the operator enters in these labels, the program SETUP.EXE automatically assembles the appropriate data and program files to carry out job planning for this particular workstation configuration. The manner in which some

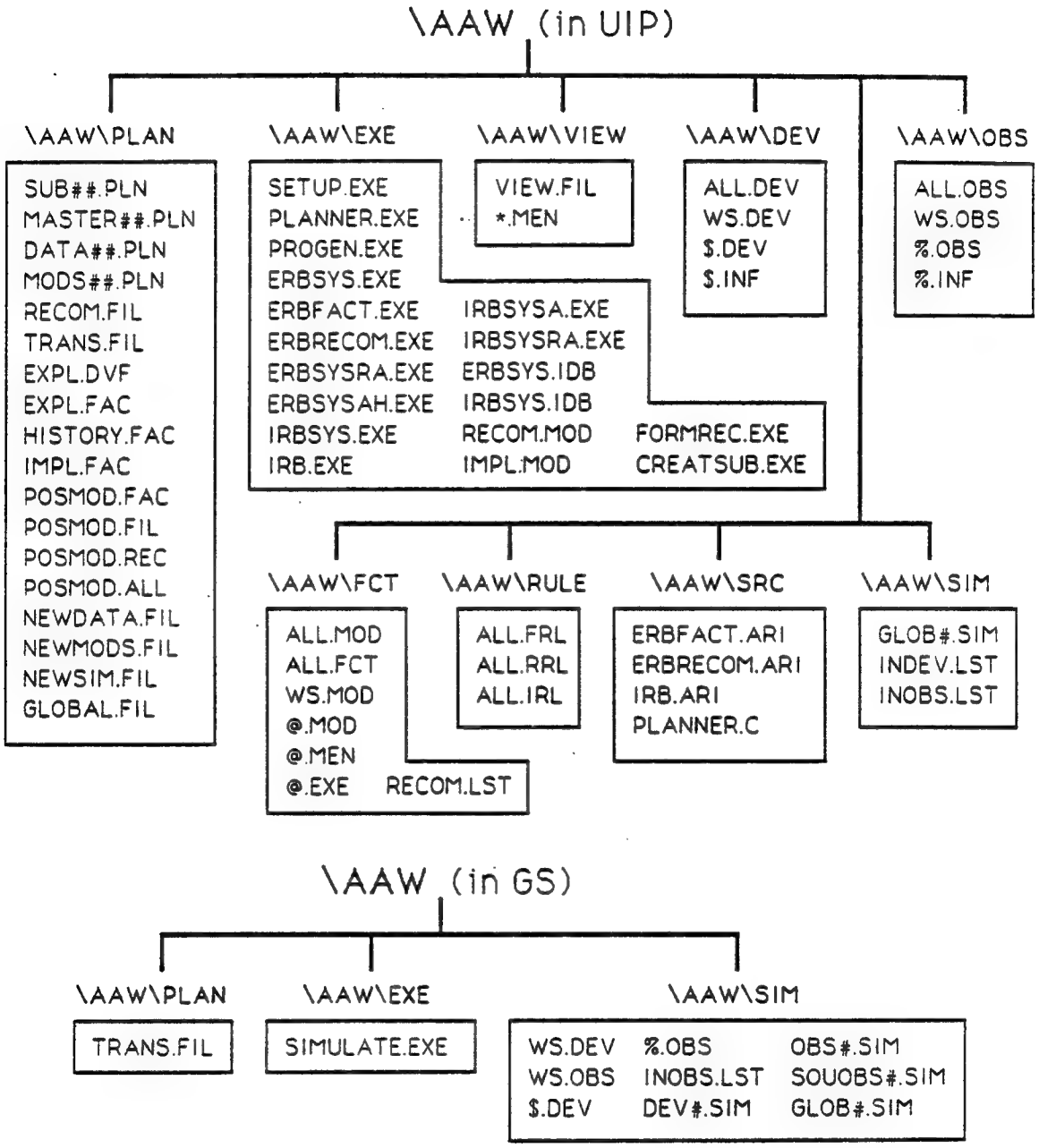


Figure G.1. ODCAP File Organization

ODCAP files are affected by SETUP.EXE is discussed in Chapter IV.

The specification of initial device settings is not performed by SETUP.EXE but must be carried out by the operator using a text editor. The data file \AAW\SIM\INDEV.LST contains the initial device settings and can be changed to correspond with device settings that the operator selects. The manner in which an operator specifies initial device data in INDEV.LST is explained by example. Figure G.2 depicts the sample contents of a typical INDEV.LST file (these contents are used for the WIDGET assembly example described in Chapter V). Only devices which are "movable" or possess device variables or "switches" (see the "modeling" section in Chapter IV for definitions of all these terms) need be included in INDEV.LST. The data for one device is separated from another by a line containing the character set "++++++++++". The first device listed in Figure G.2 is a movable device labeled BOLTDRV which possesses no device variables or switches. Movable devices are specified with three pieces of information: the name of the device to which the movable device is attached (called the "coupling" device), the coordinate frame label for the coupling device to which the movable device is attached, and the matrix transformation which defines the geometrical relationship between the movable and coupling device frames. Each transformation is specified as four columns of values (three values per column); the first three columns define

```
++++++++++
BOLTDRV
TOOLRACK
TOOLRACK
      0.0000       0.0000       1.0000    -262.5000
      0.0000       1.0000       0.0000      37.5000
     -1.0000       0.0000       0.0000      25.0000
++++++++++
GRIPPER
TOOLRACK
TOOLRACK
      0.0000       0.0000       1.0000    -100.0000
      0.0000      -1.0000       0.0000     150.0000
      1.0000       0.0000       0.0000      25.0000
***V
GRPSTATE = OPEN
***S
++++++++++
ROBOT
t1 =  -40.0
t2 =   40.0
d3 =   50.0
t4 =   30.0
***V
++++++++++
INCONV
ib =   0.0
***V
++++++++++
TOOLGRP
TGRPSTAT = OFF
***S
```

Figure G.2. Initial Device Data File

the orientation matrix while the fourth column represents the position vector. For BOLTDRV, the label TOOLRACK is indicated as the name of its coupling device and as the name of the coordinate frame to which it is "coupled".

The next device contained in INDEV.LST (devices can be listed in any order the operator desires) is the device named GRIPPER. GRIPPER is a movable device which possesses device variables and switches. All the device variables for GRIPPER are determined by the state of its switches, so the operator does not have to explicitly indicate in INDEV.LST the device variable settings. The character set "***V" must be entered (beneath the movable device specifications) whenever a device has variables even if they are all switch-affected. The device GRIPPER possesses a single switch named GRPSTATE which is given the initial state of OPEN as indicated in Figure G.2. The character set "***S" is used to indicate that all the switch settings for the device have been specified. Device data should always be entered in the order demonstrated by GRIPPER, with movable device data given first, then device variable settings, followed by switch settings.

The third device labeled ROBOT requires specification of the initial values of its four device variables: "t1", "t2", "d3", and "t4". ROBOT possesses no switches so these variables cannot be switch-affected and must be explicitly defined. The units of millimeters and degrees are implicitly assumed for device variables corresponding to lengths

and angles respectively. The last two devices INCONV and TOOLGRP have a single device variable and device switch, respectively, defined as shown in Figure G.2.

Objects which are to enter the workstation by means of source devices are characterized by the information in the data file \AAW\SIM\INOBS.LST. Appendix F should be consulted for information on object sourcing. The sample contents of a typical INOBS.LST file are provided in Figure G.3, involving objects attached to two source devices INCONV and BOLTFDR. The information pertaining to each object group (four groups for INCONV and three for BOLTFDR) is preceded by a line containing the character "*". The number of objects contained within each group is specified, followed by the name and transformation values corresponding to each object in the group. The transformation values define the geometric relationship between the source device and the object which is to be sourced and are designated in the same manner as movable device transformations. The character set "**" is used to indicate that no additional object groups are attached to a particular source device.

The operator can begin job planning once he/she is satisfied with the workstation configuration and the initial device and object information in INDEV.LST and INOBS.LST. The program files \AAW\EXE\SIMULATE.EXE (in the computer GS) and \AAW\EXE\PLANNER.EXE (in the UIP) should be executed by the operator (in the order given) when he/she wants to begin

```
INCONV
*
 2
TOPCOMP
       0.0000      -1.0000       0.0000     150.0000
       0.0000       0.0000      -1.0000     -50.0000
       1.0000       0.0000       0.0000     880.0000
BOTCOMP
       0.0000       1.0000       0.0000     390.0000
      -1.0000       0.0000       0.0000     -50.0000
       0.0000       0.0000       1.0000     830.0000
*
 1
TOPCOMP
       0.0000      -1.0000       0.0000     290.0000
       0.0000       0.0000      -1.0000     -50.0000
       1.0000       0.0000       0.0000     590.0000
*
 2
BOTCOMP
       0.0000       1.0000       0.0000     200.0000
      -1.0000       0.0000       0.0000     -50.0000
       0.0000       0.0000       1.0000     255.0000
TOPCOMP
       0.0000      -1.0000       0.0000     450.0000
       0.0000       0.0000      -1.0000     -50.0000
       1.0000       0.0000       0.0000     265.0000
*
 1
BOTCOMP
       0.0000       1.0000       0.0000     290.0000
      -1.0000       0.0000       0.0000     -50.0000
       0.0000       0.0000       1.0000     -30.0000
**
BOLTFDR
*
 1
BOLT
       0.0000       0.0000       1.0000       0.0000
       0.0000      -1.0000       0.0000      10.0000
       1.0000       0.0000       0.0000      75.0000
*
 1
BOLT
       0.0000       0.0000       1.0000       0.0000
       0.0000      -1.0000       0.0000      60.0000
       1.0000       0.0000       0.0000      75.0000
*
 1
BOLT
       0.0000       0.0000       1.0000       0.0000
       0.0000      -1.0000       0.0000     110.0000
       1.0000       0.0000       0.0000      75.0000
**
```

Figure G.3. Incoming Object Data File

job planning. ODCAP will immediately inquire from the operator the value of the plan identification number. If the number provided by the operator does not correspond to existing plan files in the \PLAN subdirectory (namely the four files possessing the extension ".PLN" which have "##" in their names to signify their plan identification number), ODCAP assumes a regular planning mode. If the number does correspond with that of existing plan files, the operator is asked by ODCAP which of the three available planning modes: replanning, simulation, and re-simulation, should be assumed. The four planning modes are defined in Chapter IV in the section which addresses the planning algorithm. All four modes require the information in the file INOBS.LST but only the regular planning and replanning modes require the initial device settings defined in INDEV.LST. The other two modes acquire initial device data from the introductory data section (which is located in the file \AAW\PLAN\DATA##.PLN).

A brief description is provided for what the operator can expect regarding each of the four planning modes. ODCAP's user interface informs the operator whenever the planning algorithm needs information form him/her during the course of the planning session. The simulation mode only requires that the operator change the viewing perspective (if he/she desires) of the graphics simulation before the task corresponding to each plan module is performed (in simulation). For the re-simulation mode, the operator has no choice concerning which tasks are performed (as is the

case in the simulation mode) yet must provide ODCAP with information related to how these tasks should be performed. Regular planning involves the specification by ODCAP of recommended tasks for plan inclusion. The operator selects tasks from these recommendations which achieve the objectives that he/she desires of the job plan. The replanning mode combines elements of the regular planning and simulation modes. When placed in a replanning mode, ODCAP asks the operator if he/she wants to insert a new module into the plan following the simulated execution of every explicit module in the existent plan. If the operator desires that the existent plan be augmented, ODCAP continues with the recommendation of suitable tasks just as it does during regular planning.

The operator can end the planning session for the regular planning and replanning modes once a task is successfully performed. The session terminates automatically for the simulation and re-simulation modes after all the tasks associated with the existent plan are performed. For the planning and replanning modes, ODCAP will further inquire from the operator if he/she desires the creation of process-level subplans from the Maser Plan. If the operator wishes to generate the language code programs which implement the plan, the process-level subplans first must be created. The program \AAW\EXE\PROGEN.EXE (in the UIP) is executed by the operator to generate the language code programs from the subplans. The operator need only supply this program with

the plan identification number (when prompted); the program will automatically access the subplans (located in the plan file \AAW\PLAN\SUB##.PLN) and assemble the desired control programs.

Appendix H

AAW Communication and Synchronization

Two RS-232 serial interfaces are used to link together the Apparel Assembly Workstation (AAW) control computers; one interface is placed between the computers termed the System Supervisor (SS) and the Vision Controller (VC) while the other is situated between the SS and the Robot Controller (RC). Communication software which implements message passing between the computers along the serial links has been created in Microsoft C and "V+". This appendix examines the operation of this software without delving into programming specifics. A means of synchronizing the AAW computers has also been developed which operates independently from the serial communication software. The operation of this synchronization method is presented.

The serial links between the three AAW computers are connected to the ports labeled COM1 and COM2 on the SS and the VC and the port named USER1 on the RC as shown in Figure H.1. The serial ports are configured to accept a baud rate of 19200 which is the fastest rate attainable in RS-232 communication. The Microsoft C functions (contained in the control code downloaded to the SS and the VC) and the "V+" routines (contained in the control code downloaded to the RC) which implement communication are "multithreaded" with the functions and routines which implement workstation

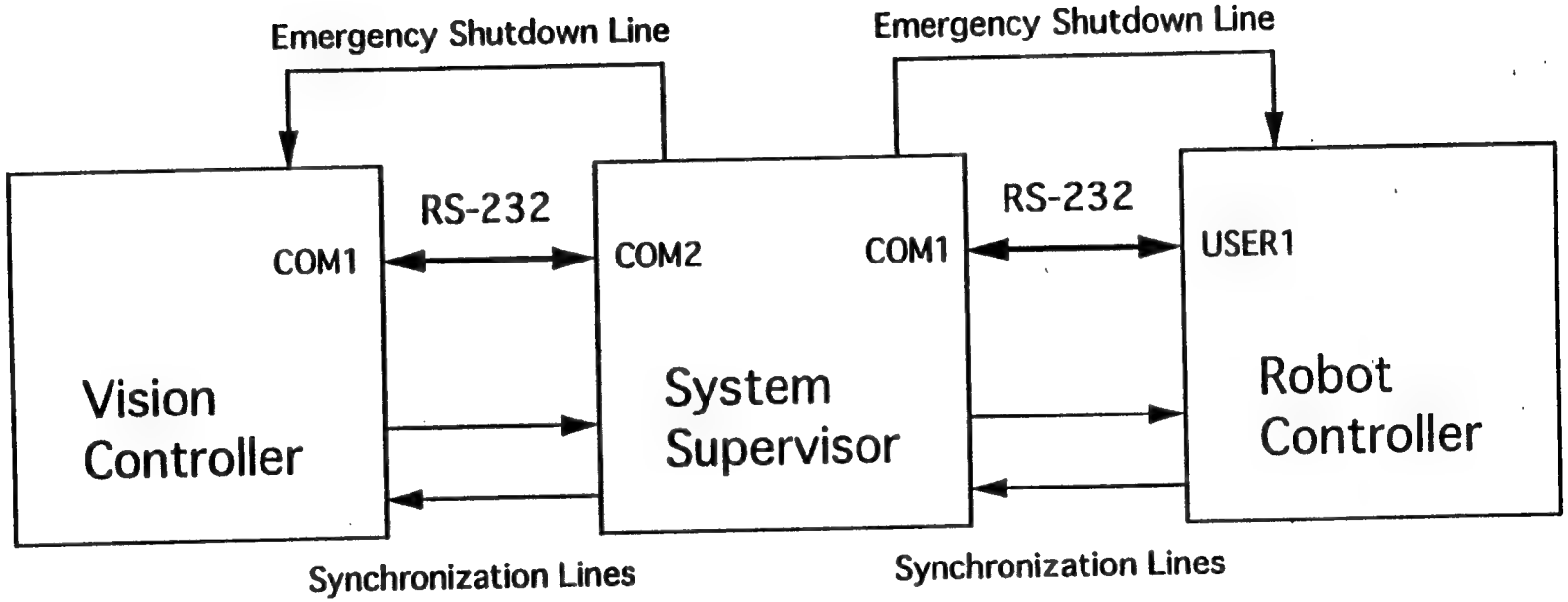


Figure H.1. AAW Computer Connections

activities other than communication. Multithreading requires that the execution of all non-communication functions and routines is suspended while the communication functions and routines are executed. After a brief time interval (about 55 milliseconds for the "C" software) has elapsed, the execution of the suspended functions and routines is resumed and the execution of the communication functions and routines is suspended. This suspension/resumption process is repeated continually throughout the performance of a job plan.

Messages for AAW communication are comprised into character sets of fixed length known as "packets". A packet is made up of 32 characters (an amount considered optimal by Shin and Epstein [48]) arranged in a structured format. Each message packet is divided into five character groupings defined as follows:

1	2	3	4	5
XXX	XX	XX	XXXXX	XXX...X

where "X" denotes a single character and the numbers identify the character groupings. The first character grouping holds a three-digit message identification number (such as "001") used to track the messages which are exchanged by the AAW computers. Every packet constituting a given message will possess the same three-digit identification number. Messages are differentiated by their points of departure and arrival. For instance, the numbering of messages passed from the SS to the RC is done separately from that of

messages sent in the opposite direction, from the RC to the SS.

The second and third character groupings indicate the packet number for the message and the total number of packets in the message, respectively. A single message can be composed of from one to 99 packets but typically would consist of less than five packets. A five-character code specifying the message's type is contained in the fourth character grouping. The remaining twenty characters of each packet store the data which is to be communicated; packets which have data comprising less than twenty characters must be padded with extra characters so that the packet's character length is held constant (at a value of 32). Message packets are queued using file buffers, after they are received by a serial port or before they are sent out from a serial port. The "C" functions and "V+" routines which perform the "reading" and "writing" of messages access message data from the file buffers and not directly from the serial ports. Transmission/reception rates of 1500 characters per second were achieved between the AAW computers.

A method for synchronizing the AAW computers using handshaking over digital I/O lines has been devised. Two lines which transmit/receive TTL signals are connected between the AAW computers as illustrated in Figure H.1. The TTL status (either 5 volts which is termed "high" or 0 volts which is "low") is reversed along one line when a computer desires synchronization with another computer. The second

computer responds, whenever it is ready, by changing the status of the other TTL line. When the first computer determines that the status of the second TTL line has been reversed, the computers can be considered synchronized.

Another critical feature associated with the real-time control of workstation hardware is the ability to shut the system down if "something" goes wrong. An emergency shut-down system has been devised whose "panic button" is linked to an interrupt in the SS software. When the operator presses the panic button, the software interrupt is immedi-ately triggered resulting in the execution of an emergency shutdown routine. This routine brings all workstation devices to a safe stop by signaling the AAW computers with TTL lines extending from the SS to the RC and to the VC, as indicated in Figure H.1. The emergency shutdown routine gives priority to stopping equipment already "in motion" such as the Adept robot since its imminent collision with some device or object may be the reason for the operator's instigation of the shutdown.

Appendix I

Program Generation

Program generation occurs after all planning operations have been performed by ODCAP and by a workstation operator, resulting in the creation of a task-level Master Plan and a set of process-level subplans. This appendix examines how Planning Level code conversion software, referred to as the program generator, transforms ODCAP modules and functions into programming language code. Complete details regarding to the operation of the program generator are not provided but a general understanding of code conversion is conveyed using "C" language examples. The program generator is contained within the executable file \AAW\EXE\PROGEN.EXE; the manner by which the operator uses this software for program generation is discussed in Appendix G.

Each ODCAP subplan consists of a list of functions corresponding to the processes which are performed when the subplan is implemented on the appropriate workstation control computer. These functions are derived from job plan modules (every module's function sequence is developed by a system programmer) and are arranged automatically into subplans at the conclusion of a planning session. Program generation concerns the conversion of ODCAP functions and their associated data structures into matching language code statements. The code statements are arranged such that specific control programs are produced for each workstation

computer in the programming language (i.e. "C" or "V+") used for the on-line operation of that computer. A single language code control program is created for each subplan. The construction of these control programs is divided along task-level and process-level hierarchical lines, in similar fashion to ODCAP job plans.

A language code control program is composed of three sections: the code function definition section, the code variable declaration section, and the main body section. The terms "code function" and "code variable" refer to a procedural entity (task-level) and a data structure (task- and process-level), respectively, corresponding to a specific programming language; they should not be confused with ODCAP functions and data structures. A code function is created when a sequence of subplan functions, derived from a single plan module, are converted into language code statements. One code function must be created for each control program for every module listed in the Master Plan. The code function definition section contains code definitions for all the code functions used in a particular control program. Code variables at a task-level are termed main body variables and at a process-level are identified as local variables. Main body and local variables correspond to ODCAP task- and process-level variable-structures respectively. The code variable declaration section comprises declaration statements for all the main body variables used in a control program. The main body section is made from

the language code statements which "call up" the code functions and effect their implementation on workstation hardware when the control programs are executed (after compilation) by the workstation computers.

Language code conversion is carried out by the program generator for every ODCAP function within a given subplan. The generator determines how each function (along with its input and output data structures) is transformed into language code statements based on conversion information supplied by a system programmer. This information is categorized by module name or by function name (dependent on which function is undergoing conversion) and is incorporated in files in the \FCT subdirectory of the UIP (see Figure 4.1 on page 71 for a listing of ODCAP files).

Figure I.1 includes a small portion of the subplan listings created for the AAW collar turning and pressing job presented in Chapter VI. The SS functions ("SS" refers to a particular AAW control computer) displayed in the figure serve to demonstrate program generation. Functions I0385 through I0387 all correspond to one particular application of the implicit module **intro_dev_var_info()()** in the Master Plan developed for the AAW collar turning and pressing job (this Master Plan is shown in Figures 6.7 and 6.8 on pages 148 and 149). This module application (identified as I013 in the Master Plan) corresponds to a computational task which inputs the contents of two data structures (into the memory of the SS) containing information on the variable

FCT #	VC LISTING	SS LISTING	RC LISTING
.	.	.	.
.	.	.	.
.	.	.	.
I0385		enter_modpart (INDEVVAR, 0000) ()	
I0386		intro_devvarinfo (PRESSER) (numdevvar, devvarinfo)	
I0387		exit_modpart (INDEVVAR, 0000, numdevvar, devvarinfo) (ndevvarPRESSER, devvariPRESSER)	
.	.	.	.
.	.	.	.
.	.	.	.
E0001		enter_modpart (RPTCOUNT, 0011) ()	
E0002		intro_sginte () (countval)	
E0003		creat_counter_info () (ival)	
.	.	.	.
.	.	.	.
.	.	.	.

Figure I.1. Partial Subplan Listings

settings of the device with the FINITE-STATE label of PRESSER.

Functions (such as I0385, I0386, and I0387) which correspond to a single module application always include at least one pair of functions named enter_modpart()() and exit_modpart()(). These two functions serve two purposes; the first of which is demarcating the ODCAP functions which are to be added (following code conversion) to the code function definition corresponding to the module in question. Secondly, the arguments contained in the input and output lists of enter_modpart()() and exit_modpart()() determine how task-level data (main body variables) is transformed into process-level data (local variables) and vice versa. The first two arguments of enter_modpart()() provide the shortened name of the module to which the functions between enter_modpart()() and exit_modpart()() correspond and a four-digit number used by the program generator to determine how code conversion information should be retrieved from ODCAP data files. The enter_modpart()() (I0385) example function shown in Figure I.1 has two input arguments, INDEVVAR and 0000. The program generator utilizes the name INDEVVAR (indicating the **intro_dev_var_info()()** module) to retrieve the module information file \AAW\FCT\INDEVVAR.MOD, in which all code conversion information is held for the functions enter_modpart()() (I0385) and exit_modpart()() (I0387).

The remaining input arguments of enter_modpart()() represent the task-level data structures (main body variables) which comprise the module's input. Each one of these arguments possesses a process-level counterpart in the output list of enter_modpart()(). These process-level data structures represent local variables used within the code function definition. When the code function is executed, the contents of the main body variables are passed to their local variable counterparts situated in the code function definition. The arguments of the exit_modpart()() function determine, similarly, which local variables are to have their contents passed back to main body variables following the code function's execution. The INDEVVAR module has two main body variables in the output list of exit_modpart()() (I0387) namely "ndevvarPRESSER" and "devvariPRESSER". The contents of these variables are obtained from the local variables "numdevvar" and "devvarinfo", respectively, when the code function corresponding to INDEVVAR completes its on-line execution.

The program generator retrieves code conversion information from module information files when generating language code for the functions enter_modpart()() and exit_modpart()(). The code conversion information in module files is categorized by the workstation computer to which it corresponds. The first box in Figure I.2 shows all the code conversion information (taken from the ODCAP data file \AAW\FCT\INDEVVAR.MOD) associated with the INDEVVAR module

```
**** SS CODE INFO 0 ****
MINPARGS:
---
FOUTARGS:
---
FINPARGS:
intesg
struID
dev_variable_info
---
MOUTARGS:
intesg
struID
dev_variable_info
---
DECL:
int ^mout01;
struct dev_variable_info ^mout02[MAXNUMDEVVAR];
---
MAIN:
?FST 01
intro_dev_var_infoF---S(&^mout01,
                        ^mout02);
---
HEAD:
?FST 01
void intro_dev_var_infoF---S(^finp01p,
                             ^finp02)
int *^finp01p;
struct dev_variable_info ^finp02[MAXNUMDEVVAR];
{
?SHF 02
---
TAIL:
?SHF -2
}
---
```

```
***** SS intro_sginte *****
INPARGS:
---
OUTARGS:
intesg
---
LOCALS:
int *^fout01p;
---
BODY:
*^fout01p = #fout01#;
---
```

Figure I.2. Code Conversion Information

and the SS computer. Conversion information is grouped into different categories with each category represented by some label. (Labels are denoted by capitalized letters.) Category labels are included as part of the code conversion information (with each label followed by a colon); conversion information corresponding to each category is placed between the category label and the character set "---". The first four categories for the INDEVVAR example of Figure I.2 are labeled MINPARGS, FOUTARGS, FINPARGS, and MOUTARGS. The information in these categories defines how the main body and local variables represented by the input/output arguments of enter_modpart()() (I0385) and exit_modpart()() (I0387) are to be classified in the language code of the SS. Variable classifications, such as "intesg" under the category FINPARGS, need only be understood by the system programmer who develops the code conversion information.

The fifth category shown in Figure I.2 for INDEVVAR is titled DECL, providing the language code statements which must be added to the code variable declaration section. These statements (as are all code statements) are altered such that the variable names used in the ODCAP function listings are integrated into the code statement syntax. The character "^" and the six characters which follow it represent a single "character code block" which must be replaced by one of the ODCAP function arguments. For example, the character set "^mout01" should be replaced with the first argument in the output list of exit_modpart()(). The code

statements under the DECL category for module INDEVVAR are converted into the following:

```
int ndevvarPRESSER;
struct dev_variable_info devvariPRESSER[MAXNUMDEVVAR];
```

These statements are added directly to the code variable declaration section.

The category MAIN contains the language code statements which must be added to the main body section. The first statement included in the INDEVVAR example is "?FST 01". Any code statement beginning with the character "?" denotes a format statement which instructs the program generator about the conversion of subsequent code statements. Format statements are identified by the three characters following the "?". Many different types of format statements are available for use by the programmers who create the language code conversion information files. The format statement "?FST 01" indicates to the program generator that the character set "F---S" situated in the code statement following the format statement should be replaced by the first FINITE-STATE value connected to the module INDEVVAR. The code statements under the MAIN category for module INDEVVAR are converted into:

```
intro_dev_var_infoPRESSER(&ndevvarPRESSER,
                          devvariPRESSER);
```

These statements trigger the implementation of the code function corresponding to INDEVVAR.

Two remaining categories, termed HEAD and TAIL, contain the code statements corresponding to the start and end

statements of the code function definition. Every code function definition is assembled after the program generator completes conversion of the code statements pertaining to function exit_modpart()(). The program generator adds code function definition statements to the code function definition section and also adds main body statements and code variable declaration statements to their respective sections.

All ODCAP functions other than enter_modpart()() and exit_modpart()() are processed in a different manner by the program generator. Code conversion information for these functions is placed in the single data file \AAW\FCT\ALL.FCT and is referenced by function name as well as by the computer label to which the function applies. Conversion of a typical ODCAP function is demonstrated using the SS function intro_sginte()() (E0002) shown in the subplan listings of Figure I.1. The code conversion information for the function is reproduced in the second box in Figure I.2.

Every ODCAP function (other than enter_modpart()() and exit_modpart()()) has its code conversion information divided into four categories: INPARGS, OUTARGS, LOCALS, and BODY. The first two categories provide the language code classifications of the input/output arguments of the ODCAP function. The function intro_sginte()() (E0002) has only one output data item named "countval" which is classified as "intesg". The category LOCALS contains the code statements which declare the function's local variables within the code

function definition to which the function belongs. The code statements associated with intro_sginte()() (E0002) would be to added to the code function definition corresponding to the module RPTCOUNT (see the enter_modpart()() function situated directly above intro_sginte()() in Figure I.1).

The category BODY includes the code statements which implement the ODCAP function when the appropriate control program is executed. Code conversion of the single BODY statement connected to intro_sginte()() (see Figure I.2) requires that the program generator access information from the introductory data section, contained in the data file \AAW\PLAN\DATA##.PLN. The pair of "#" characters incorporated in this statement enclose special code characters which the program generator uses (along with the function identification number and the computer label) to determine the location of a particular data value in the introductory data section. The data value is retrieved and is written explicitly into the converted code statement. The code statement for the BODY category of intro_sginte()() is converted into:

```
*countvalp = 2;
```

The character set "countval" is taken from the first output argument of the function (E0002) while the value "2" is obtained from the introductory data section.

Functions which are not situated between the functions enter_modpart()() and exit_modpart()() have their converted BODY code statements added directly to the main body section

while their converted LOCALS statements are appended to the code variable declaration section. After the last subplan function is converted, the code function definition section is combined with the main body and code variable declaration sections so that a viable workstation control program is produced. The program generator eliminates all repetitions of local and main body variable declaration statements to ensure that no single variable is ever declared twice either within a code function definition or within the main body. Code function definitions other than those corresponding to plan modules may have to be added to a control program if they are not explicitly defined in the code conversion information files. The workstation operator makes uses of a control program produced by by ODCAP's program generator just as he/she would utilize any source code developed to control on-line workstation activities.

LIST OF REFERENCES

1. Compton, W. D., Ed., Design and Analysis of Integrated Manufacturing Systems, National Academy Press, Washington, 1988, pp. 79-91.

2. Shapiro, S. F., Ed., "Robotic Systems Learn Through Experience," Computer Design, November 1988, pp. 54-68.

3. Smith, M. F., Software Prototyping: Adaption, Practice and Management, McGraw-Hill Book Company, London, 1991, pp. 1-9.

4. Bourne, D. A. and M. S. Fox, "Autonomous Manufacturing: Automating the Job Shop," Computer, September 1984, pp. 76-86.

5. Albus, J. S., Brains, Behavior, and Robotics, BYTE Books, Peterborough, New Hampshire, 1981, pp. 101-138, 261-299.

6. Saridis, G. N., "Intelligent Robotic Control," IEEE Transactions on Automatic Control, Vol. AC-28, No. 5, May 1983, pp. 547-557.

7. Saridis, G. N., "Toward the Realization of Intelligent Controls," Proceedings of the IEEE, Vol. 67, No. 8, August 1979, pp. 1115-1133.

8. Meystel, A., "Intelligent Control in Robotics," Journal of Robotic Systems, Vol. 5, No. 4, August 1988, pp. 269-308.

9. Norcross, R. J., "A Control Structure for Multi-Tasking Workstations," Proceedings of the IEEE International Conference on Robotics and Automation, Philadelphia, April 1988, pp. 1133-1135.

10. Albus, J. S., C. R. McLean, A. J. Barbera, and M. L. Fitzgerald, "Hierarchical Control for Robots in an Automated Factory," Proceedings of the 13th International Symposium on Industrial Robots and Robots 7, Chicago, April 1983, pp. 13-29-13-43.

11. Acar L., "A Knowledge-Rich Hierarchical Controller for a Redundant Manipulator," Proceedings of the IEEE International Conference on Systems Engineering, Pittsburgh, August 1990, pp. 9-12.

12. Schalkoff, R. J., Artificial Intelligence: An Engineering Approach, McGraw-Hill, Inc., Hightstown, New Jersey, 1990, pp. 424-538.

13. Brooks, R. A., "Symbolic Error Analysis and Robot Planning," The International Journal Of Robotics Research, Vol. 1, No. 4, Winter 1982, pp. 29-68.

14. Xia, X. D. and G. A. Bekey, "Integrating Robotic Assembly Planning and Scheduling: A Two-Dimensional View," Proceedings of the IEEE International Conference on Robotics and Automation, Cincinnati, May 1990, pp. 1956-1961.

15. Gevarter, W. B., Intelligent Machines: An Introductory Perspective of Artificial Intelligence and Robotics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985, pp. 67-81.

16. Hutchinson, S. A. and A. C. Kak, "Extending the Classical AI Planning Paradigm to Robotic Assembly Planning," Proceedings of the IEEE International Conference on Robotics and Automation, Cincinnati, May 1990, pp. 182-189.

17. Lu, S. C.-Y., "Knowledge Processing for Engineering Automation," Proceedings of the 15th Conference on Production Research and Technology, Berkeley, California, January 1989, pp. 455-467.

18. Gevarter, W. B., "Introduction to Artificial Intelligence," Chemical Engineering Progress, September 1987, pp. 21-37.

19. Prajoux, R., R. Sobek, A. Laporte, and R. Chatila, "A Robot System Utilizing Task-Specific Planning in a Blocks-World Assembly Experiment," Proceedings of the 10th International Symposium on Industrial Robots, Milan, Italy, March 1980, pp. 281-292.

20. Paul, R. P., H. F. Durrant-Whyte, and M. Mintz, "A Robust, Distributed Sensor and Actuation Robot Control System," Proceedings of the Third International Symposium on Robotics Research, Gouvieux, France, October 1985, pp. 93-100.

21. Pang, G. K. H., "Automatic Off-Line Programming of Robots," Proceedings of the Winter Annual Meeting of the ASME, DSC-Vol. 16, San Francisco, December 1989, pp. 41-47.

22. Laugier, C. and J. Pertin-Troccaz, "SHARP: A System for Automatic Programming of Manipulation Robots," Proceedings of the Third International Symposium on Robotics Research, Gouvieux, France, October 1985, pp. 125-132.

23. Xia, X. and G. A. Bekey, "SROMA: An Adaptive Scheduler for Robotic Assembly Systems," Proceedings of the IEEE International Conference on Robotics and Automation, Philadelphia, April 1988, pp. 1282-1287.

24. Caudill, R. J., T. L. Chiang, E. S. Pound, and T. V. King, "ESSOP: An Expert Support System for Off-Line Programming of Industrial Robots," Proceedings of the Winter Annual Meeting of the ASME, Anaheim, California, PED-Vol. 24, December 1986, pp. 211-218.

25. Dufay, B. and J.-C. Latombe, "An Approach to Automatic Robot Programming Based on Inductive Learning," The International Journal of Robotics Research, Vol. 3, No. 4, Winter 1984, pp. 3-20.

26. Sheridan, T. B. and W. R. Ferrell, Man-Machine Systems: Information, Control, and Decision Models of Human Performance, The MIT Press, Cambridge, Massachusetts, 1974, pp. 1-19, 93-107.

27. Groover, M. P., Automation, Production Systems, and Computer Integrated Manufacturing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987, pp. 309-335.

28. Taylor, R. H., J. U. Korein, G. Maier, and L. F. Durfee, "A General-Purpose Control Architecture for Programmable Automation Research," Proceedings of the Third International Symposium on Robotics Research, Gouvieux, France, October 1985, pp. 165-173.

29. Lozano-Perez, T., "Robot Programming," Proceedings of the IEEE, Vol. 71, No. 7, July 1983, pp. 821-841.

30. Summers, P. D. and D. D. Grossman, "XPROBE: An Experimental System for Programming Robots by Example," The International Journal of Robotics Research, Vol. 3, No. 1, Spring 1984, pp. 25-39.

31. Rembold, U., "Programming of Industrial Robots: Today and in the Future," Languages for Sensor-Based Control in Robotics, NATO ASI Series, Vol. F29, 1987, pp. 3-23.

32. Shen, H. C. and A. K. C. Wong, "A Model for Robot Programming and Control," Languages for Sensor-Based Control in Robotics, NATO ASI Series, Vol. F29, 1987, pp. 45-65.

33. Johnson, D. G. and J. J. Hill, "Sensor Level Programming: A New Software System for Improved Control of a Sensory Industrial Robot," Proceedings of the 5th International Conference on Robot Vision and Sensory Controls, Amsterdam, October 1985, pp. 383-392.

34. Hill, J. J., D. C. Burgess, and A. Pugh, "The Vision-Guided Assembly of High-Power Semiconductor Diodes," Proceedings of the 14th International Symposium on Industrial Robots, Gothenburg, Sweden, October 1984, pp. 449-459.

35. Osada, M., "Development of Visual Feedback Robots, Designed to Assemble Complex Automotive Components," ASME Proceedings of the USA-Japan Symposium on Flexible Automation, Minneapolis, July 1988, pp. 823-828.

36. Nevins, J. L. and D. E. Whitney, Ed., Concurrent Design of Products and Processes: A Strategy for the Next Generation in Manufacturing, McGraw-Hill Publishing Company, New York, 1989, pp. 197-230, 279-313.

37. Boothroyd, G., C. Poli, and L. E. Murch, Automatic Assembly, Marcel Dekker, Inc., New York, 1982, pp. 255-297.

38. Gaetan, M., "Robots: Their Potential in the Apparel Industry," Bobbin, August 1981, pp. 83-92.

39. McPherson, E. M., "Sewing Industry Research," Bobbin, September 1982, pp. 121-128.

40. Taylor, P. M., A. J. Wilkinson, G. E. Taylor, M. B. Gunner, and G. S. Palmer, "Automated Fabric Handling Problems and Techniques," Proceedings of the IEEE International Conference on Systems Engineering, Pittsburgh, August 1990, pp. 367-370.

41. Taylor, P. M. and G. E. Taylor, "Progress Towards Automated Garment Manufacture," Proceedings of the NATO Advanced Research Workshop on Sensory Robotics for the Handling of Limp Materials, Il Ciocco, Italy, October 1988, pp. 97-109.

42. Bernardon, E. and T. S. Kondoleon, "Real Time Robotic Control for Apparel Manufacturing," Conference Proceedings for Robots 9, Detroit, June 1985, pp. 4-46-4-66.

43. Gershon, D. and I. Porat, "Vision Servo Control of a Robotic Sewing System," Proceedings of the IEEE International Conference on Robotics and Automation, Philadelphia, April 1988, pp. 1830-1835.

44. Nakamura, T., T. Arai, Y. Tanaka, M. Satoh, and Y. Imazu, "Trajectory Generation of Redundant Manipulator For 3-D Sewing System," ASME Proceedings of the USA-Japan Symposium on Flexible Automation, Minneapolis, July 1988, pp. 35-40.

45. Torgerson, E. and F. W. Paul, "Vision-Guided Robotic Fabric Manipulation for Apparel Manufacturing," IEEE Control Systems Magazine, Volume 8, No. 1, February 1988, pp. 14-20.

46. Paul, F. W., E. Torgerson, S. Avigdor, D. Cultice, A. Gopalswamy, and K. Subba-Rao, "A Hierarchical System for Robot-Assisted Shirt Collar Processing," Proceedings of the IEEE International Conference on Systems Engineering, Pittsburgh, August 1990, pp. 378-382.

47. Fitzgerald, M. L. and A. J. Barbera, "A Low-Level Control Interface for Robot Manipulators," Robotics & Computer-Integrated Manufacturing, Vol. 2, No. 3/4, 1985, pp. 201-213.

48. Shin, K. G. and M. E. Epstein, "Communication Primitives for a Distributed Multi-Robot System," Proceedings of the IEEE Conference on Robotics and Automation, St. Louis, March 1985, pp. 910-917.

49. Harmon, S. Y., "Implementation of Complex Robot Subsystems on Distributed Computing Resources," Machine Intelligence and Knowledge Engineering for Robotic Applications, NATO ASI Series, Vol. F33, 1987, pp. 407-435.

50. Stewart, D. B., D. E. Schmitz, and P. K. Khosla, "Implementing Real-Time Robotic Systems Using CHIMERA II," Proceedings of the IEEE International Conference on Robotics and Automation, Cincinnati, May 1990, pp. 598-603.

51. Jones, A., E. Barkmeyer, and W. Davis, "Issues in the Design and Implementation of a System Architecture for Computer Integrated Manufacturing," International Journal of Computer Integrated Manufacturing, Vol. 2, No. 2, 1989, pp. 65-76.

52. Fielding, P. J., F. DiCesare, and G. Goldbogen, "Error Recovery in Automated Manufacturing through the Augmentation of Programmed Processes," Journal of Robotic Systems, Vol. 5, No. 4, August 1988, pp. 337-362.

53. Lam, R. K., N. S. Pollard, and R. S. Desai, "Studies in Knowledge-Based Diagnosis of Failures in Robotic Assembly," Proceedings of the IEEE International Conference on Systems Engineering, Pittsburgh, August 1990, pp. 60-65.

54. Smith, R. E. and M. Gini, "Robot Tracking and Control Issues in an Intelligent Error Recovery System," Proceedings of the IEEE International Conference on Robotics and Automation, San Francisco, April 1986, pp. 1070-1075.

55. Cox, I. J. and N. H. Gehani, "Exception Handling in Robotics," Computer, March 1989, pp. 43-49.

56. Gini, G., M. Gini, and M. Somalvico, "Program Abstraction and Error Correction in Intelligent Robots," Proceedings of the 10th International Symposium on Industrial Robots, Milan, Italy, March 1980, pp. 101-108.

57. Taylor, G. E. and P. M. Taylor, "Dynamic Error Probability Vectors: A Framework for Sensory Decision Making," Proceedings of the IEEE International Conference on Robotics and Automation, Philadelphia, April 1988, pp. 1096-1100.

58. Sharir, M., "Algorithmic Motion Planning in Robotics," Computer, March 1989, pp. 9-19.

59. Ramamritham, K. and M. Arbib, "Distributed Control and Scheduling for Robots," Proceedings of the 15th Conference on Production Research and Technology, Berkeley, California, January 1989, pp. 133-141.

60. Barbera, A. J., M. L. Fitzgerald, and J. S. Albus, "Concepts for a Real-Time Sensory-Interactive Control System Architecture," Proceedings of the 14th Southeastern Symposium of System Theory, Blacksburg, Virginia, April 1982, pp. 121-126.

61. Valavanis, K. P. and G. N. Saridis, "Analytical Design of Intelligent Machines," Proceedings of the 1st Symposium on Robot Control of the International Federation of Automatic Control, Barcelona, Spain, November 1985, pp. 139-144.

62. Monckton, S. P. and R. W. Toogood, "A Compact Prolog Implementation of an Assembly Graph World Model," Proceedings of the Winter Annual Meeting of the ASME, DSC-Vol. 16, San Francisco, December 1989, pp. 49-54.

63. Jain, A. and M. Donath, "Knowledge Representation Model for Robot Based Assembly Planning," Proceedings of the American Control Conference, Minneapolis, June 1987, pp. 181-187.

64. Shneier, M., E. Kent, J. Albus, P. Mansbach, M. Nashman, L. Palombo, W. Rutkowski, and T. Wheatley, "Robot Sensing for a Hierarchical Control System," Proceedings of the 13th International Symposium on Industrial Robots and Robots 7, Chicago, April 1983, pp. 14-50-14-66.

65. Paul, R. P., Robot Manipulators: Mathematics, Programming and Control, MIT Press, Cambridge, Massachusetts, 1981.

66. Fikes, R. and T. Kehler, "The Role of Frame-Based Representation in Reasoning," Communications of the ACM, Vol. 28, No. 9, September 1985, pp. 904-920.

67. Avigdor, S., Development of an Automated Double Point Collar Turning Machine, Master of Engineering Project Report, Mechanical Engineering Department, Clemson University, Clemson, South Carolina, December 1991.

68. Subba-Rao, K., Vision-Assisted Edge Alignment for the Automated Pressing of Two-Dimensional Apparel Components, MS. Thesis, Mechanical Engineering Department, Clemson University, Clemson, South Carolina, August 1991.

69. Gopalswamy, A., Design and Control of a Robot End-Effector for Three Dimensional Manipulation of Multiple-Ply Apparel Workpieces, MS. Thesis, Mechanical Engineering Department, Clemson University, Clemson, South Carolina, December 1990.

70. Cultice, D. R., Three-Dimensional Vision Sensing for Fabric Manipulation, MS. Thesis, Mechanical Engineering Department, Clemson University, Clemson, South Carolina, August 1991.

71. Shimano, B. E., C. C. Geschke, C. H. Spalding III, and P. G. Smith, "A Robot Programming System Incorporating Real-Time and Supervisory Control: VAL-II," Conference Proceedings for Robots 8, Detroit, June 1984, pp. 20-103-20-119.

72. Takanashi, N., H. Ikeda, T. Horiguchi, and H. Fukuchi, "Hierarchical Robot Sensors Application in Assembly Tasks," Proceedings of the 15th International Symposium on Industrial Robots, Tokyo, September 1985, pp. 829-836.

73. Clocksin, W. F. and C. S. Mellish, Programming in Prolog, Springer-Verlag, New York, 1981.